DOT/FAA/AR-03/77

Office of Aviation Research
Washington, D.C. 20591

Commercial Off-The-Shelf Real-Time Operating System and Architectural Considerations

February 2004

Final Report

U.S. Department of Transportation
Federal Aviation Administration

NOTICE

This report is available at the Federal Aviation Administration William J. Hughes Technical Center’s Full-Text Technical Reports page: actlibrary.act.faa.gov in Adobe Acrobat portable document format (PDF).

Technical Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
DOT/FAA/AR-03/77		

4. Title and Subtitle	5. Report Date
COMMERCIAL OFF-THE-SHELF REAL-TIME OPERATING SYSTEM AND ARCHITECTURAL CONSIDERATIONS	February 2004
	6. Performing Organization Code

7. Author(s)	8. Performing Organization Report No.
Jim Krodel	

9. Performing Organization Name and Address	10. Work Unit No. (TRAIS)
United Technologies Research Center 411 Silver Lane East Hartford, CT 06108	
	11. Contract or Grant No. DTFA03-01-P-10129

12. Sponsoring Agency Name and Address	13. Type of Report and Period Covered
U.S. Department of Transportation Federal Aviation Administration Office of Aviation Research Washington, DC 20591	May 2002 – May 2003
	14. Sponsoring Agency Code AIR-120

15. Supplementary Notes

The FAA William J. Hughes Technical Center COTR was Charles Kilgore.

16. Abstract

This report investigates the safety aspects of using commercial-off-the-shelf (COTS) real-time operating system (RTOS) software in aviation systems. Because of the complexity and unknown integrity of many COTS RTOSs, there are a number of concerns regarding their use in aircraft systems, as they may potentially affect aircraft safety. Previous Federal Aviation Administration-sponsored research in this area identified COTS RTOS as a potential focus area for COTS in aviation software applications. This follow-on work considers the use of partition-supporting COTS RTOS in aviation systems.

This report specifically studies the hardware and software architectural issues of embedded aviation software systems, with a particular focus on systems with the multilevel criticality partitioning provided by COTS RTOS. As a representative example, this report also discusses the PowerPC microprocessor architecture and how its architecture relates to the safety aspects of COTS RTOS running on this processor.

17. Key Words	18. Distribution Statement
Software, Commercial Off-the-Shelf, DO-178B, IMA, Partitioning, APEX, ARINC 653	This document is available to the public through the National Technical Information Service (NTIS), Springfield, Virginia 22161.

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	58	

ACKNOWLEDGMENTS

The author wishes to express his appreciation for the support of George Romanski of Verocel, Inc. who provided significant contributions to this research and to Robert Thornton of United Technologies Research Center. They provided excellent technical advice throughout the development and preparation of this report.

The author is also very grateful to other engineers and researchers in several aerospace companies, but in particular, to Mark Jenkinson and Jeff Schmidt of Hamilton Sundstrand and Vivek Halwan of United Technologies Research Center.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ACRONYMS

ADS-B	Automated Dependent Surveillance – Broadcast
APEX	Application Executive (ARINC 653)
API	Application Programming Interface
ARINC	Aeronautical Radio, Inc.
ARINC 653	Avionics Application Software Standard Interface by ARINC
BAT	Block Address Table
BSP	Board Support Package
CC	Common Criteria
CM	Configuration Management
COTS	Commercial Off-The-Shelf
CPU	Central Processing Unit
DMA	Deadline Monotonic Analysis
DO-178B	Software Considerations in Airborne Systems and Equipment Certification document by RTCA, Inc.
DO-254	Design Assurance Guidance for Airborne Electronic Hardware document by RTCA, Inc.
EIOEIO	Enforce In Order Execution of I/O
FIFO	First In First Out

GPS	Global Positioning System
I/O	Input / Output
IMA	Integrated Modular Avionics
ISO	International Organization for Standardization
LIFO	Last In First Out
MMU	Memory Management Unit
MOS	Module Operating System
NSA	National Security Agency
OEA	Operating Environment Architecture
OS	Operating System
POS	Partition Operating System
POSIX	Portable Operating System Interface
PSAC	Plan for Software Aspects of Certification
RISC	Reduced Instruction Set Computer
RTOS	Real-Time Operating System
SEU	Single Event Upset
SQA	Software Quality Assurance
TLB	Translation Look-aside Buffer
UISA	User Instruction Set Architecture
VEA	Virtual Environment Architecture
WCET	Worst-Case Execution Time
XML	Extended Markup Language

EXECUTIVE SUMMARY

The purpose of this report is to document the investigation of the safety aspects of using commercial-off-the-shelf (COTS) real-time operating system (RTOS) software in aviation systems and the architectural strategies that may be necessary.

The Federal Aviation Administration (FAA) is concerned with the isolation and fault containment provided by the partitioning function within some RTOSs. As a result of the industry movement to put more applications on a single computing platform, the complexity of hardware and software has increased, and analysis of adequate protection is challenging. In particular, one needs to consider the subject of adequate protection with respect to determining the ability of a system to establish independence between functions and fault containment of functions with lesser criticality assessments. Partitioned systems may also provide a vehicle for reduced recertification cost if the area of change is contained to a particular partition, has no affect on the memory allocations of other partitions, and does not change process timing or major frame scheduling.

The FAA sponsored this research in the area of COTS software to identify technical and safety issues regarding its uses as well as to identify areas to be addressed in future certification policy and guidance. In Phases 1 and 2 of the COTS avionics software study, COTS RTOS was identified as a potential focus area for COTS in aviation software applications. The Phase 3 study began an in-depth study into the considerations of using COTS RTOSs in aviation systems by focusing on the characteristics of COTS RTOSs. This report, Phase 4, continues and concludes the in-depth study by focusing on related architectural strategies.

The procedures and approach taken during this research included obtaining past experience with COTS RTOS architectures, reviewing literature available in the public domain, reviewing product information, subcontracting a portion of the research to Verocel, Inc., and applying a stress test plan.

This report specifically studies the hardware and software architectural issues of embedded aviation software systems, with a particular focus on systems with multilevel criticality partitioning provided by some COTS RTOSs. Although this study focuses on embedded aviation software systems, much of its content is applicable to other aviation or safety domains. Partitioned systems are seen as a natural vehicle for protecting various levels of software as defined in DO-178B and are candidates for supporting integrated modular avionics (IMA) systems. Although partitioning can be applied within a single processor or across a distributed system, this report focuses on the single processor application. Many discussion points in this report can be extrapolated across such a distributed system.

RTOSs that are commercially developed may not have the rigor required for operation in today's aviation systems. Typical safety-critical systems have used well-established and well-understood processors, but a trend has emerged toward an IMA approach that takes advantage of the powerful features of complex processors such as the PowerPC. This report provides a discussion of IMA systems. It is understood that this term can be viewed as a complex multicabinet, multirack system; however for the scope of this report, IMA is viewed as a

partitioned system with multiple applications and its associated RTOS component that services these applications.

Because of the potential lack of development and verification rigor of a commercial RTOS, a COTS RTOS should consider the availability of DO-178B life cycle artifacts, the ability to meet safety-critical requirements, and the adaptability of the COTS RTOS to the chosen processor.

Many system implications can arise from COTS RTOS attributes. Board support packages, memory models, and caching mechanisms are a few of those considered. The PowerPC has emerged as a candidate for many of the complex embedded computer systems of today and is the basis of several systems with multilevel criticality partitioning. Its features are used as a basis of analysis for this study, especially with respect to partition-supporting RTOS architectures.

In this study, it was found that memory in partitioned systems will need to be managed, including static allocation of code, maximum stack sizes, and sizes and locations fixed for memory heap. A variety of other partition-supporting RTOS considerations are further discussed in this report such as cache jitter, worst-case execution time, health monitoring, and system integration. This study also revealed that RTOS security features might conflict with safety features.

An examination of robustness testing for COTS RTOSs showed that a large portion of robustness testing could exist as part of an RTOS's normal functional test, if it is included at all in the requirements and design of the system. From a stress test point of view, the process of stress testing an RTOS could raise system design issues not previously considered. Target linking and testing revealed that because a RTOS supports multipartitioning, a vehicle for specifying which software goes into which partition is needed. For highly critical applications, careful scrutiny of the partitioning allocation mechanism is needed because the linker generates and integrates code into the target system.

1. INTRODUCTION.

1.1 PURPOSE.

The purpose of this report is to investigate the safety aspects of using commercial-off-the-shelf (COTS) real-time operating system (RTOS) software in aviation systems. A trend to use commercially available RTOS software in aviation systems has emerged due to the perceived cost and schedule savings associated with using COTS components. Because of the complexity and unknown integrity of many COTS RTOSs, there are a number of concerns regarding their use in aircraft systems, as they may potentially affect aircraft safety. The Federal Aviation Administration (FAA) sponsored this research in the area of COTS software to identify technical and safety issues regarding its uses as well as to identify areas to be addressed in future certification policy and guidance. A previous COTS avionics software study [1] identified COTS RTOS as a potential focus area for COTS in aviation software applications. This follow-on work studied the use of COTS RTOS in aviation systems.

This report specifically studies the hardware and software architectural issues of embedded aviation software systems, with a particular focus on systems with multilevel criticality partitioning provided by the COTS RTOS. Although this study focuses on embedded aviation software systems, much of its content is applicable to other aviation or safety domains. Partitioned systems are seen as a natural vehicle for protecting various levels of software as defined in DO-178B and are candidates for supporting integrated modular avionics (IMA) systems. Partitioned systems may also provide a vehicle for reduced recertification cost if the area of change is contained to a particular partition, has no affect on the memory allocations of other partitions, and does not change the process timing or major frame scheduling of any process in the system or the system as a whole. Although partitioning can be applied within a single processor or across a distributed system, this report focuses on the single processor application. Many discussion points in this report can be extrapolated across such a distributed system.

A variety of other partition-supporting RTOS considerations are summarized below and will be further discussed in this report:

- Board Support Package (BSP). The BSP is software that isolates the RTOS from the target computer. It permits the RTOS to reside on various hardware architectures. The BSP initializes the processor, devices, and memory; performs various memory checks; and so on. Once initialization is complete, the BSP can still function to perform low-level cache manipulations. Much of this code can only operate in privileged mode and works hand-in-hand with the RTOS. In a partitioned system, placement of the BSP device drivers can vary with regard to the overall partitioned architecture. While the COTS RTOS may be developed as a general-purpose operating system usable on potentially many different processors, the BSP is typically highly customized for each specific application, target computer configuration, and set of resources.

- Cache Jitter. Cache memory is a global resource that is typically shared between partitions. The information in cache may contain page address values, data values, and code specific to a partition. Time is compromised, if cache flushing is needed when partitions change via the round-robin scheduler or some other scheduler. Beyond this is the possibility that a partition could potentially change the internal state of the cache, which perhaps induces a failure of another partition.

- Worst-Case Execution Time (WCET). For any system requiring determinism, a WCET analysis becomes a much more complex task for a pipelined processor such as the PowerPC. Building a robust model of the PowerPC is complex, and verifying this model can be difficult. On the other hand, too simple a model can provide such inaccuracies or long-timing predictions that the choice of using such a complex microprocessor is questioned. Several approaches are taken; the most promising are those that not only build a model of relative fidelity but also validate the model with actual system-timing assessments properly representing the application's operating environment. The complexity of the model required is proportional to the processor hardware attributes and scheduling methods employed. For instance, a model that uses pipelining, cache, and resource-blocking mechanisms would need to be more complex than models that only require cache.

- Health Monitoring. Health monitoring is intrinsic to the Avionics Application Standard Interface by ARINC (ARINC 653) specification that covers both the space and time domains. The specification identifies a number of error codes that must be detected and handled. An error can be handled within a process, in a partition, or in the health monitoring module. Health monitoring code can be used to contain the errors, to substitute alternate actions, or simply to record the errors in an error log. Non-ARINC 653-compliant designs must address health monitoring as well. Noncompliant designs should be able to address how the ARINC 653-compliant health monitors are handled in the noncompliant design. In either case, the need for, or the identification of, additional health monitors should be considered. The system design, hardware design, operating system design, or partitioning design may justify the need for health monitors and actions not called out by the ARINC 653 specification.

- System Integration. The final IMA system must have an overall system integrator that allocates and integrates the resources of the various applications on the IMA. The resources used by an application must be strictly controlled because indiscriminate resource usage could affect the behavior of another partition.

The reasons for using a partitioned system can vary. Although this report considers, in several areas, partitioning for software-level isolation, there are many others, including the simple separation of functions.

As a representative example, this report discusses the PowerPC microprocessor architecture, and how its architecture relates to the safety aspects of using a partition-supporting COTS RTOS running on the PowerPC processor.

Note that this is a research report. The information contained will be used for input to policy and guidance but does not itself constitute FAA policy or guidance.

In addition, ARINC 653 terminology is used in this report, and ARINC 653 was being updated at the time this report was written.

1.2 DOCUMENT OVERVIEW.

Section 1 provides the purpose and scope of this report, and it also presents clarification of some of the terms in the report.

Section 2 provides traceability to several related reports. It presents a basis for the area under study in this report. It briefly discusses the COTS acquisition process and summarizes the architecture of the PowerPC, which is used as a basis for this study.

Section 3 discusses standards that may be employed and focuses on approaches to space and time partitioning.

Section 4 builds upon the RTOS and partitioned multiple application system architectures presented in section 3. Specific considerations for using partition-supporting RTOSs in IMA systems are considered, i.e., board support packages, cache jitter, background partition, data flow between partitions, worst-case execution time analysis, scheduling within partitions, integration, and RTOS libraries.

Section 5 was not initially planned for inclusion in this report, but in the course of this research, COTS RTOS implementations of National Security Administration security-based functions were discovered to have potential impact on system safety.

Section 6, a previous phase of this study (Phase 3), explored the COTS RTOS domain and its safety implications. The Phase 3 report suggested and supplied a robustness test plan for a partition-supporting RTOS as potential template for an RTOS vulnerability analysis and test. The plan was provided to an applicant of such a system and its findings are reported.

Section 7 discusses the results and recommendations for COTS RTOS, their architecture, and areas of further study.

Section 8 lists the references used in this report.

Section 9 lists related documentation.

2. BACKGROUND.

This report summarizes Phase 4 of a four-phase study regarding COTS components. Phases 1 and 2 discussed issues regarding the use of both software and hardware electronic COTS components in aviation systems, which resulted in two reports [1 and 2]. The third phase took a

detailed look into the safety and certification issues of using a COTS RTOS in aviation applications.

The COTS electronics report from Phases 1 and 2 provided findings about the state of the industry relative to the design objectives identified in guidance document DO-254 with a focus on the implications of the use of COTS electronic hardware components in safety-critical airborne systems. The use of complex electronic hardware components in airborne systems poses a challenge to meeting safety requirements because, for complex components, complete verification is, at best, very difficult, and, at worst, not achievable.

The COTS software report from Phases 1 and 2 provides a snapshot of portions of industry domains related to safety. Avionics, nuclear, medical, space, and elevator domain information was surveyed. Key industry COTS components were identified and potential alternate methods for verifying the applicability to the avionics domain application of a COTS component were studied. Real-time operating systems and communications software emerged as key eminent COTS technology offerings in the aviation community.

Phase 3 of this study further explored the COTS RTOS domain and its safety implications [3]. RTOS attributes were detailed and their safety-related properties were discussed along with considerations to address when integrating a COTS RTOS with an application in an aviation system. A stress or robustness test was presented as an example for the basis of an RTOS vulnerability analysis.

Phase 4 is the basis for this report, as described in section 1.1, and builds upon the previous studies. The purpose of this fourth phase is to understand and document the safety implications of a partitioned COTS RTOS and its integrated architecture features.

Some RTOSs may have data to support DO-178B and some may not. Further, some COTS vendors helped or provided certification support activities for such things as BSP and packaging the data. The level of support and supporting data may be software-level relevant.

2.1 COTS RTOS SELECTION PROCESS.

Because the selected COTS RTOS will replace a significant portion of the system and its associated design, the COTS RTOS selection process, and its associated implications, must be well understood. The process of selecting a COTS RTOS has many implications at the requirements and design phase of the system under development. It has been demonstrated that although the design phase may be 5% of the overall cost of developing a system, the design phase can affect over 70% of the overall product cost of a system and in-service maintenance cost [4].

Please note that some issues presented below are germane to COTS software in general and some are particular to COTS RTOSs. The following discussion does not propose solutions to the COTS RTOS selection process, but simply poses a set of considerations for COTS software selection and specifically for COTS RTOS selection and its associated development and target environment considerations.

2.1.1 General Considerations in COTS Software Selection.

Many COTS vendors have committed, or have plans to commit, to providing COTS products made for the aviation sector. Because these products are targets for aviation systems, their pedigree and applicability to the specific application being developed must be carefully considered. A new set of thought processes is required when a COTS product is introduced into the system; one quite different from more traditional aviation software development approaches. A few of the questions to be asked for COTS systems, in general, follow [5]:

- What affect does this COTS product have on the development life cycle of the system?
- Is the product compatible with other components being developed?
- Is this product flexible, or does it put constraints on the system under development?
- How complex is the product, and is there sufficient data to support the required level of criticality?
- How will the possibility of obsolescence with the COTS product be handled?
- What happens if the COTS vendor goes out of business?
- What kind of problem-reporting system is in place for the COTS product to support continued airworthiness?
- What assurance is there that the COTS product will not be changed without the user's knowledge?

Beyond these considerations for the general COTS software product, a COTS RTOS has many more considerations simply because a RTOS is a system-capability enabler. Essentially, it is the heartbeat of the aviation system.

2.1.2 Target-Specific COTS RTOS Selection in Airborne Systems.

The RTOS provides a set of services to many objects in the target system. It communicates to hardware devices, schedules tasks, sends and receives messages to and from tasks, controls functional priorities, and in some cases, provides partitioning of system functions to permit different safety criticality levels to coexist. As such, a much deeper set of considerations is needed, as the target system for which the COTS RTOS is to be inserted has the potential to cause catastrophic aircraft behavior. The questions to be asked of the RTOS, as a minimum, are [1]

- What are the potential hazards that can be posed by the RTOS itself?
- What affect does the RTOS have on the overall certification effort, and will this be detailed in the Plan for Software Aspects of Certification?

- If information commensurate with meeting DO-178B is lacking, what alternate methods should be considered to show evidence of compliance to the objectives of DO-178B?

- Has the COTS RTOS been held to standards compliance, and what is the evidence supporting this?

- What kind of data is available to support the COTS usage in aviation products (e.g., does a certification package exist?)?

- Which Configuration Management (CM) and Software Quality Assurance (SQA) approaches were used in the COTS RTOS development, and which CM and SQA approaches will be used for these products within the target system in its deployment?

2.1.3 COTS RTOS Selection Considerations in Complex Target Systems.

Software began to emerge as a key capability in aviation systems many years ago, creating a need for software to govern the system resources. Early on, this was in the form of simple foreground/background executives and runtime kernels that integrated task execution to permit required system operations. The technology was simple, and its scope was limited. However, as embedded software systems have grown, so have their complexity. RTOSs have become tightly integrated into target systems, resulting in the exploitation of complex hardware. Correspondingly, the complex environments for development of RTOSs have also given rise to further considerations in the use of COTS RTOS. The rise in these attributes of RTOSs has created yet another set of considerations (as a minimum):

- What is the RTOS timing performance; e.g., latencies, thread switch jitter, scheduling schemes, and priority levels?

- In what language and compiler was the RTOS developed, and is it deterministic?

- Does the compiler take advantage of processor attributes of out-of-order execution, cache, and pipelines?

- Was the RTOS developed for use with a specific processor or family?

- Is the source code available for scrutiny by higher criticality level application developers?

- Which hardware resources does the RTOS affect?

- What affect does software used to isolate the RTOS from the target computer (e.g., BSP) have on the system certifiability?

- How does the RTOS handle the following functions, and does it do so in a deterministic manner? (1) task handling; (2) memory management; and (3) interrupts.

- How does the RTOS use memory, such as internal and external cache?

- Which tools are available for analyzing the RTOSs performance, and can these tools verify deterministic behavior?

- Does the RTOS have security capabilities that conflict with the systems safety objectives?

- Does the central processing unit (CPU) have an internal memory management unit (MMU), and does it support partitioning?

This research will study many of these complex COTS RTOS considerations to establish an understanding of the potential basis for certification approval of such systems.

2.2 SUMMARY OF POWERPC ARCHITECTURE.

In order to form a basis for discussing advanced microprocessors, a popular microprocessor architecture was selected for study. The PowerPC is being proposed on a number of airborne partition-supporting systems. The 7XX family, in particular, was studied and permits partitioning via its robust memory management features. Other PowerPC families may have some different features and attributes that require different evaluations for safety considerations with RTOSs.

The following section provides high-level background information on the architecture of the PowerPC family of microprocessors and will focus on only those features that play a role in RTOS interactions.

The PowerPC architecture jointly developed by Motorola, IBM, and Apple Computer was designed to provide a single architecture for a variety of processor environments [6]. Because of the designed-in flexibility and scalability, features of the design may be present that may require risk mitigation activities from a system safety point of view. This subsection studies the use of the PowerPC with RTOSs and some architecture considerations of RTOS and will neither study nor present in detail any risk mitigation activities specific to the PowerPC.

2.2.1 Key PowerPC Features.

For scalability purposes, the PowerPC does not define individual signals or the bus protocol. Designers can choose to implement architecturally defined features in hardware or software. The basic architecture of the PowerPC permits what is called Harvard Architecture, separating instructions and data memory space. It also permits a unified cache where instructions and data are combined.

The architecture includes information on the instruction set, a programming model that defines the register set and memory conventions, and the memory model itself, which defines the address space of segments, pages, and blocks. It provides User-mode instructions to control the on-chip cache with such operations as store, flush, and data invalidate. It also provides Supervisor-mode instructions.

For defining the processor state, the Machine State Register includes the state of power management, endian byte ordering, interrupt enabling, privilege levels, machine check, floating-point exceptions, and instruction and data address translation.

The PowerPC architecture also includes an MMU that converts the effective address or logical address used by memory access or branch instructions to the appropriate physical address.

In essence, because the PowerPC is designed to be scalable and flexible, the PowerPC initialization and setup is important in understanding not only how a particular PowerPC-based system is designed, but also for understanding its capabilities with regards to the RTOS that runs on it.

2.2.2 Modes of Operation.

There are three basic PowerPC architecture levels. First is the User Instruction Set Architecture, which defines the base User-mode instruction set, User-mode registers, data types, floating-point memory conventions, and exception model. Second is the Virtual Environment Architecture (VEA), which allows multiple devices to access memory, defines the time base facility, and defines the cache model and instructions. Third is the Operating Environment Architecture (OEA) that defines the memory management model, the Supervisor-mode registers, synchronization requirements, and the exception model. The features of the OEA are accessible to Supervisor-mode applications only, typically operating systems.

The PowerPC is designed for flexibility. Because of this, portions of the architecture are not defined by the chip design itself. The PowerPC does not define individual signals or the bus protocol. For example, the OEA allows each implementation to determine the signal or signals that trigger the machine check exception. In addition, the PowerPC architecture itself does not define the cache size, structure, replacement algorithm, or mechanism used for maintaining cache coherency. For this reason, it is important to understand the applications model and how that model is used for each specific PowerPC application design.

2.2.3 Supervisor/User Mode.

Two processor levels of privilege exist: Supervisor mode, which is typically used by the operating system (OS), and User mode, which can be used by both the application and operating system. With these features, the OS can control the application environment, providing virtual memory and protection for the OS and memory partitions and other critical machine resources, as appropriate.

2.2.4 Memory Management and Addressing.

The MMU of the PowerPC performs address translation for load and store instructions. The MMU also translates cache instructions and external control instructions. This translation mechanism is defined in terms of the segment descriptors and page tables used by the PowerPC to locate the effective physical address mapping. The definition of the segment and page table data structure provides significant flexibility for performance and can be used to store the

segment or page table information on-chip. In addition, Translation Lookaside Buffers (TLBs) are commonly used to keep recently used page address translations on-chip.

Segment descriptors are used to generate interim virtual addresses, and may reside on-chip, or as segment table entries in off-chip memory. The Block Address Translation (BAT) mechanism is a software-controlled array that stores the available block address translations on-chip. BAT array entries are implemented as pairs of BAT registers, accessible as Supervisor Special-Purpose Registers.

The PowerPC provides for three basic address translation mechanisms:

- Page address translation: translates the page frame address for a 4-Kbyte page.
- Block address translation: translates the block number for blocks ranging from 128 Kbytes to 256 Mbytes.
- Real addressing mode translation: effective address is identical to the physical address.

PowerPC processor versions within the 7XX family may differ in the specific hardware used to implement the MMU model on the OEA. Thus the MMU, with the exception of the processing mechanism, is capable of providing the necessary operating system support to implement a paged virtual memory environment and enable enforced protection of designated memory areas. For this reason, the PowerPC is a candidate for performing partitioning on a single-chip CPU.

2.2.5 Pipelining.

Pipelining is a method of efficiently using processor resources such that independent activities are performed in a parallel manner. A limitation with the technique is its execution time prediction, which can vary based on the data presented to the pipelined architecture. The PowerPC is a basic Reduced Instruction Set Computer (RISC) architecture, designed to facilitate the use of pipelining and parallel execution to maximize instruction throughput. The designer even has the luxury of specifying the execution of load and store operations in the integer unit instead of the dedicated load and store unit.

This flexibility provides a tremendous advantage to the user. The user can prescribe instruction-fetching mechanisms, how the instructions are decoded and dispatched, and how results are written back to memory. In addition, dispatch and write-back may occur in order or out of order. When this code is written to manipulate data that is only visible to the processor, the sequencing imposed by the processor optimizes the throughput but does not compromise it. If data is shared through external access, i.e., by an external memory mapped input/output (I/O) device, then the data access sequence becomes critical. Information may be in cache, or it may be enroute to be saved in memory using internal hardware buffers not visible to the user. The hardware specifications describe instruction sequences that must be used to ensure that the buffers are written in an order that ensures coherency of the memory model. On the PowerPC, instructions such as Enforce In Order Execution of I/O (EIOEIO) and SYNCH are used to control these memory access sequences.

In general, the user will not take direct advantage of the pipeline. When compiler optimization is switched on, high-level language compilers become aware of the pipeline and may elect to take advantage of it. By analyzing the instruction sequence, a compiler may remove a conditional branch instruction and replace it with a set of arithmetic operations. The motivation is to remove conditional branch instructions because they stall the pipeline. A full pipeline will partially execute a number of instructions on each clock cycle. The effect is that instructions that actually may require several machine clock ticks to execute appear to be fully executed on each clock cycle.

This issue may raise a problem with coverage analysis. A verification technique that uses code insertion to trace the execution of multiple source-level conditions will be mapped very differently if the trace code is removed. With tracing, the code will be mapped to multiple conditions; without tracing, the code may be mapped onto arithmetic or logical operations. If the code is verified at DO-178B level A, coverage analysis of the RTOS will require careful analysis.

2.2.6 Cache.

The flexibility provided in the PowerPC also extends to the cache. The VEA and OEA architectures define aspects of cache implementation. The PowerPC allows control of several memory access modes on a page or block basis, including write-back/write-through mode, caching-inhibited mode, memory coherency, and guarded/not guarded against speculative accesses.

- Write-through mode ensures that data in the cache is updated in both the cache and a main memory location. Subsequent data load operations will use the cache copy of the data to avoid a memory read. This improves performance of the executing programs.

- Write-back mode only updates data in the cache and subsequent data reads coming from the cache. Data is loaded in the cache when accessed for the first time. The location in cache holds both the memory address and value. The location in cache is derived from the memory address, and the actual location is a function of a predefined selection of bits used in the memory address. If a cache location is occupied before a new value is loaded, the memory location must be written back. To make this more efficient, the PowerPC provides a set-associative cache mechanism. This effectively replicates the cache a number of times (2- or 4-way set associative) to improve the chances of finding the memory value in the cache. To improve the cache utilization, the hardware uses a randomizing function to ensure the different set associations are used fairly.

- Caching-inhibited mode bypasses the cache altogether and stores the data in main memory. It is important that all locations in a page are purged from the cache before changing the memory/cache access attribute from caching-allowed to caching-inhibited. Caching-inhibited mode is typically used on memory areas that are mapped so that they can be used by I/O devices.

- Memory coherency page designations use data store operations to a location in a serialized manner, with all stores to that location. This mode is useful for the implementation of memory-mapped device drivers.

- Memory coherency page designations not required necessitates that the software must ensure that the data cache is consistent with main storage before changing the mode or before allowing another device to access the area.

- Guarded page designations require that instructions and data cannot be accessed out of order.

In a strong, consistent memory model, the responsibility for access ordering is on the programmer. This hinders performance with excessive overhead, as one must wait for exclusive access to an address before updating, which causes a time loss to occur. On the other hand, write-back caching and out-of-order execution allows programmers to exploit performance benefits of weakly ordered memory accesses on the PowerPC.

Several modern processors use similar cache memory schemes to increase the speed of memory access. Using a cache memory, especially with pipelining, increases the complexity and accuracy of the WCET analysis, which may affect the measure of a system's determinism.

Cache memory is a hardware architecture mechanism whose primary purpose is to improve performance of an application running on the target processor. It accomplishes this by holding information in a local high-speed cache memory and synchronizing this information with the contents of main memory as needed. Cache memory should receive special scrutiny in a partitioned system because the cache mechanism is not aware of the address-partitioning architecture. When an application runs within one partition, it forces the CPU to load information, which is then preserved for as long as possible within the cache. Subsequent partitions may be affected by the presence of data in the cache. As cache memory is common to all partitions, the use of this resource requires careful analysis. Write-through and/or cache flushing techniques, among others, may be needed.

2.2.7 Exceptions or Interrupts.

The OEA of the PowerPC typically defines the mechanisms to implement exceptions. The mechanism permits the PowerPC to switch to the Supervisor state as a result of external signals, errors, or unusual conditions arising for instruction execution. An RTOS has the capability for handling these exceptions for the application, or it can pass the exception along for the application to handle. When an exception is raised, the processor state is saved, including specific registers. Multiple exception conditions can map to a single exception vector address, and a register associated with the exception holds more specific exception condition information. For exceptions that occur while an exception handler routine is executing, multiple exceptions become nested, and the exception handler program must save the appropriate machine state.

Exceptions can occur on a system reset, machine check, data memory access violation, instruction fetch violation, external interrupt, memory alignment, illegal instruction, privileged instruction, a decremeter, a system call, and several other processes. RTOSs vary on exception or interrupt handling, depending upon their implementation and the nature of the exception or interrupt.

When the exception arrives, the RTOS determines where it belongs. Some of the exceptions are propagated to the application for processing; for example, a divide by zero exception in an Ada program may be handled by a user-provided handler in the function in which it was raised. In an ARINC 653-compliant RTOS, some exceptions are handled by the Partition Operating System (POS), which may be in user space or system space, depending upon implementation, while some exceptions will be handled by the Module Operating System (MOS). Some of these exceptions are handled by the Health Monitor as described in section 4.7.

The RTOS should have a very clear description of its exception-handing features.

2.3 BACKGROUND ON THE PORTABLE OPERATING SYSTEM INTERFACE (POSIX) OS INTERFACE STANDARD.

RTOSs have evolved over the years into complex systems providing a variety of services. Many applications now require operating system services such as concurrent programming, device I/O, communication networks, and file systems. To create a standard set of services with consistent operations, the need for a POSIX arose. In 1990, POSIX.1 was released to provide the basis for a standard set of OS services. The standard was based on the UNIX OS, and its goal was to provide portability of applications at the source code level.

In 1993, POSIX.1b was developed to address real-time systems whose broad definition of real time in OSs was "the ability of the operating system to provide a required level of service in a bounded response time." POSIX.1b addressed real-time kernels, Ada language run-time executives, and larger OSs. These OS services attempted to achieve predictable timing behavior by addressing process scheduling, virtual memory management, real-time clocks, timers, and process synchronization. However, this definition did not address the performance needs of systems requiring high efficiency. In 1995, POSIX.1c defined C language interfaces to support multiple threads of control inside each POSIX process sharing the same address space. In 1999, POSIX.d provided additional interfaces for services such as timeouts for blocking services and spawning processes [7].

There are OSs, which by nature, are very close to the POSIX.1 specification but rarely implement all of the real-time and thread extensions of POSIX.1b and POSIX.1c. Although OS vendors may state their RTOS is POSIX conformant or compliant, this does not ensure that the RTOS has the ability to meet the deterministic behavior necessary for today's airborne systems. How the RTOS vendor developed and how the applicant uses the RTOS resources, determines the RTOS's behavior.

3 PARTITION-SUPPORTING RTOS ARCHITECTURES.

This section considers partitioning aspects of RTOSs used to support system development and implementation. Some standard approaches to partitioned RTOS architectures using ARINC 653 are considered as well as time and space partitioning and scheduling for these systems. Some RTOSs may have data to support ARINC 653 and some may not. Further, some COTS vendors are helping or providing DO-178B certification-support activities for such things as BSP and packaging the data. The level of support and supporting data may be software-level relevant.

A number of approaches to the provision of a partition-supporting RTOS are currently available, and it is expected that more will become available in the future. Proprietary RTOSs are difficult to describe in this report because their details are not published, except under nondisclosure agreements. From a practical viewpoint, this research project considered three Application Programming Interfaces (APIs), as described below.

1. The ARINC 653 specification provides an API specifically designed for a partition-supporting RTOS. Note also the part of the specification that deals with scheduling within a partition has also been proposed for use in a nonpartitioned, federated style system to preserve compatibility of application between multipartitioned and single-partition implementations. The ARINC 653 specification, also known as the Application Executive, provides bindings to the C language and the Ada language.

   The C language binding uses the International Organization for Standardization (ISO) standard, and the Ada binding was developed using Ada 83, which is upwardly compatible with Ada 95. Clearly, none of the Ada 95 features were directly used. The binding is thin, meaning the Ada binding merely provides a calling interface to the C specification. The two bindings are mappings of the abstract interface provided in the specification.

2. As noted in section 2.3, the POSIX specification is offered as a standard programming interface for RTOS use. This specification is not partitioned-based and is only applicable within a partition, because it only provides scheduling control over processes within a partition. Operations that require control of partitions or communication between partitions are outside of the POSIX specification. Although ARINC 653 was loosely based on the ideas of POSIX, the two interfaces are not compatible. Programs are not portable from one style of coding to the other without changes. An RTOS that offers both ARINC 653 and POSIX interfaces should include warnings that if the calls are mixed, there are potential dangers that the semantics of one or the other interface will be compromised.

3. The Ada language provides a set of primitive types and language features that could be used to program real-time applications. Ada compilers may provide support for these concurrency constructs by either mapping them onto an underlying RTOS or by providing a run-time system specifically developed to support the language. Applications may be run using Ada-tasking constructs and a number of language

primitives to help with concurrent real-time applications. Because the full features of the Ada language were considered too difficult to support in a safety-critical environment, a subset of the Ada language was specified using a profiling mechanism. The Ravenscar profile provides an agreed-upon language subsetting capability. This profile is accepted by the international community, including the ISO standards body, as a subset suitable for safety-critical applications (ISO/IEC TR 15942:2000).

The Ravenscar profile is not compatible with the Ada language binding of ARINC 653, and the two cannot coexist. While there are many advantages to an Ada compiler checking the legality of tasking operations at the application level, this cannot be accomplished using the Ada binding to ARINC 653.

Multitasking applications with different software level partitions require that partitions are developed and tested to those appropriate software levels' assurance. This means that failures in one partition must not affect any other partition. Multitasking architecture requirements must also apply within a partition, including, but not limited to, a deterministic priority-based pre-emptive scheduler. While ARINC 653 requires priority pre-emption, the scheduler in a partition could actually be cooperative. Added to this is the possibility that a given partition may be permitted to contain its own run-time scheduler based on another operating system altogether. This RTOS would again need to protect itself from invalid accesses and provide deterministic execution time for all services. The dichotomy can be further extended to include a separate language, which uses a compiler that includes its own run-time system in a given partition [8].

3.1 STANDARD APPROACHES.

Only the ARINC 653 APEX model will be considered in the remainder of this report, because it is the only available API standard that addresses some of the needs of IMA projects. For noncompliant ARINC 653 designs, the discussions and concerns discussed in this report may also need to be addressed.

Function protection and partitioning protection standards are not very prevalent. ARINC 653 is the most current document to detail these protection attributes. The purpose of this study is not to reflect what ARINC 653 contains. However, the report will review ARINC 653 and extract the keys to function protection and, if possible and where appropriate, present a generalized statement or safety concern.

From the user perspective, the programming model may be the application executive (APEX) specification (i.e., the API may comply with ARINC 653). The implementation may alternately be offered as a layer, which is based on the proprietary implementation offered by a specific vendor. One vendor's accommodation of ARINC 653 may be implemented with only one scheduler in the kernel providing scheduling operations for both the processes within applications, and also scheduling operations of the applications. Another vendor may use a two-level scheduler: one that manages applications and the tasks that manage their resources and a process scheduler that manages User-mode processes within an application. Other approaches are also possible.

In designing an operating system, many tradeoffs affect its behavior. For example, should I/O operations be supported by the kernel where access to the target resources are more direct, or should they be supported by the partition where the programming model is easier but where it is less efficient because data must be copied between partitions before it is output? The tradeoffs may balance latency against efficiency, and different vendors may have different approaches.

This presents a difficulty for the certification authorities because the one API model is mapped to many approaches and implementations. The different approaches should represent a uniform semantic model as defined by the ARINC 653 specification or alternate approaches that are mapped to this specification when proprietary approaches are used. This will need to be verified and demonstrated; otherwise, assumptions made by different application developers may be compromised. Some vendors have even considered reverse engineering their RTOS and applying techniques to have their RTOS ARINC 653 compliant. There may be different mechanisms and interactions, therefore, a thorough review of the approach is needed.

As recommended in Phase 3 of this study [3], potential vulnerabilities must be listed. This is comparable to providing a system safety system assessment or functional hazard analysis, but because this is an operating system, there is no system for this. Instead, the vulnerabilities should be identified, classified, and mitigated. There are times when the OS cannot mitigate the hazard. In this case, the hazard or vulnerability must be identified for the application to mitigate.

3.1.1 Programming Models as Related to Operating Systems.

ARINC 653 provides a model and an API for the implementation of applications that run on a single platform. The model's API is comparable to the features on many RTOSs. There are a number of COTS RTOSs under development, together with certification supporting materials that intend to provide implementations supporting the ARINC 653 specification. The implementation of these systems may be very different.

The concept behind a partitioned multiple-application system is to support multiple applications executing on a single computing resource and sharing I/O resources. The applications share the processor and all of the global resources. The sharing is controlled in such a way that each partition appears to be running on its own processor, which is running in periodic bursts of execution. The applications are mapped to partitions. Each application is in a separate partition that is provided with memory. The logical memory layout of a partitioned system is shown in figure 1. The remainder of section 3.1 will consider this figure as an example partitioned multiple-application system. For this example, one assumes that the processor is a PowerPC and an ARINC 653-compliant RTOS is being used.

Memory organizations are typically dictated by the hardware support for their memory protection and by the design of the RTOS. In most architectures, and in particular on the PowerPC, there is an execution state defined as Supervisor mode and another defined as User mode.

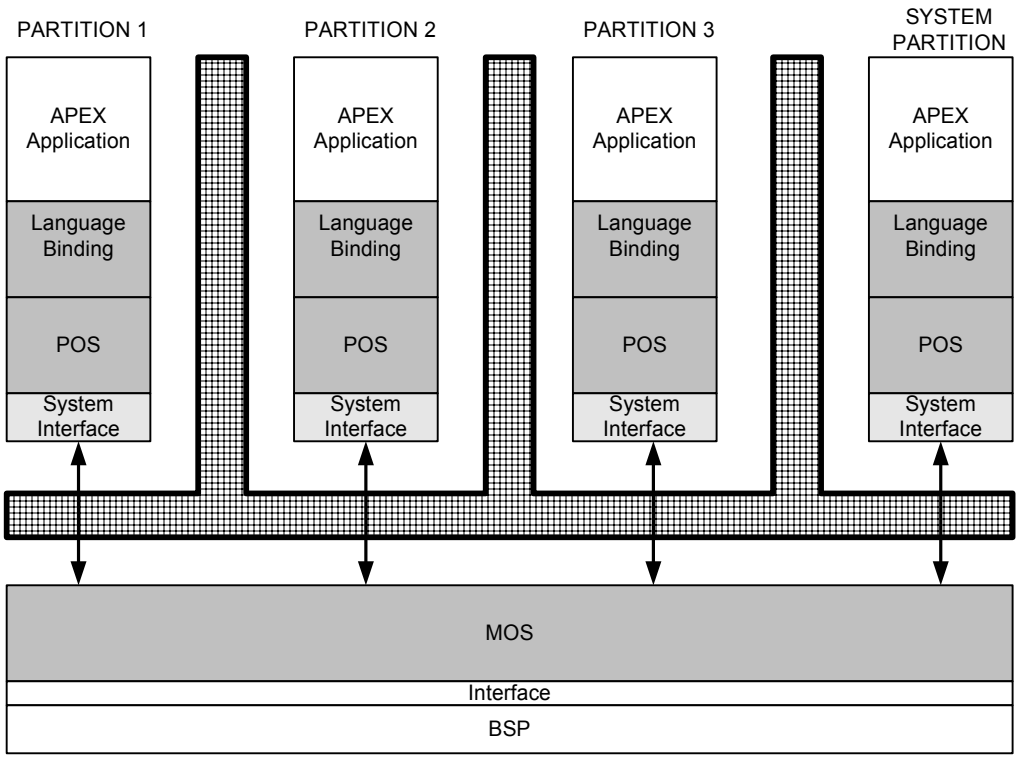

FIGURE 1. MULTIPLE-APPLICATION MEMORY LAYOUT

Figure 1 shows the applications and the POS linked together inside each partition labeled Partition 1, 2, and 3. All code and data in each partition is linked together, and this code runs at the User mode. The MOS and the components of the BSP run in the Supervisor mode. There may be an optional partition that is provided by the system supplier. This partition may have special status, and the MOS may grant additional visibility to this system partition for brief periods. This system partition could include some I/O or mode switching code.

The checkered area in figure 1 represents a protection mechanism that prohibits or strictly controls references from one partition to another and from any partition to the Supervisor mode. To prevent the applications from being completely isolated, communication mechanisms are provided that allow information to be sent in a controlled sequence between partitions and from partitions to an I/O device. These communication mechanisms are specified in ARINC 653 and offer a standard way of sending and receiving information.

3.1.2 Separation of System and User Modes.

To ensure that there is separation between the figure 1 applications and how they are managed, the hardware needs a way to maintain the status of the execution profile. This is accomplished using a status register. Setting the status register puts the computer/microprocessor into a System mode (also known as Supervisor or Privileged mode), or a User mode. The setting or clearing of the status register may only be performed when the system is in Supervisor mode. The MOS is loaded in a memory location that is available only when the computer is in

Supervisor mode. Because User-mode code cannot change the status register, applications are restricted from certain instructions that can only execute in Supervisor mode.

The status register is fundamental in separating the availability of the resources between the MOS and the partitions that include the applications. There are a number of instructions that may only be executed in Supervisor mode. These instructions may be used to setup the system so that it behaves in restricted ways. The control over memory accesses, interrupts, and timers may also be set up using privileged instructions.

3.1.3 Memory Protection Mechanisms.

The memory organization (in the figure 1 example) may be set up so that each partition appears as a contiguous logical address space. A set of address translation tables will cause the linear virtual addresses to be translated during program execution time to physical memory address locations that need not be contiguous. By organizing the memory through the translation tables, the Supervisor-mode code can control the actual memory available for each partition. In a multiple-application system, this would normally be performed by the MOS in response to some configuration tables used by the system integrator to apportion memory resources between partitions. It is possible to control memory access rights, as well as memory modes dynamically, so that read, write, or execute rights can also be controlled at run time. This is required when applications are removed or loaded from a multiple-application system during maintenance. Even though the MOS has sufficient privilege to change memory access rights dynamically, it should be avoided while in operational mode. For example, no changes in memory control access rights from the User mode should be allowed.

3.1.4 Code Protection.

By setting up suitable memory translation tables, it is possible to arrange for code to be treated specially. Certain memory regions can be set up with the execute-only memory attribute. This means that applications running at the User mode would be unable to corrupt code areas because the hardware would prevent writing to those areas. This provides a level of code protection from the User application-level code.

3.1.5 Vectoring of Interrupts.

Interrupts are asynchronous events that must be managed very carefully. They can occur at any time while the MOS or application-level code is running. It is important that they do not encroach on the time and space of a partition that has no interest in the interrupt regardless of the implementation. The source of interrupts is controlled by setting up suitable vectoring mechanisms that control the code to be executed when an interrupt arrives, preserving the interrupted execution context as well as the data describing the interrupt.

Timers provide interrupts that control scheduling events in the application through the POS as well as in the MOS. A process running in an application may delay itself for some time, or it may perform an operation that causes the process to be blocked because a resource is unavailable. The duration of this resource blocking may be limited through the addition of a

time-out parameter. As the blocking periods expire, the processes are again made available to the scheduler to select the process with the highest priority. This mechanism requires careful implementation. The following considerations apply:

1. Timer interrupts occur in System mode only. The clock cannot be made directly accessible to the application level; otherwise, the application could steal more time by adjusting the clock.

2. If the POS and MOS are in different protection levels, then a mechanism must be programmed to propagate the clock events into the application partition.

3. While a partition is active, all of its clock events must be propagated as they occur.

4. When a partition is dormant, its clock events must be stored and then transmitted as soon as the partition becomes active. This means that partition time (as seen by the application) moves in bursts relative to real time. This must be addressed when programming multiple-partitioned applications. Process times must fit in with the expected partition duration and frequency.

External I/O devices are an additional source of interrupts. Various styles of I/O devices may synchronize their transfers with the availability of data between the processor and the device itself. The recommendation in ARINC 653 is that device drivers use polled I/O; however, specific systems may resort to interrupt-driven I/O as well. The system integrator should address the frequency of interrupts because the processing performed by these interrupts consumes the processing resource upon each arrival of an interrupt. If interrupts arrive asynchronously, there is a corresponding loss of processing cycles from the partitions on which the interrupts occur. This must be estimated and considered in worst-case timing execution because it results in some jitter in the time available to the partitions.

Interrupts from external hardware events must be available, for example, from a power failure. However, these should be rare and require processing outside of the normal scheduling configuration. Such interrupts will typically be handled by the MOS, which simply pre-empts the partition that was active.

3.2 APPROACHES TO PARTITIONING SPACE AND TIME.

3.2.1 Partitioned Space Protection Mechanisms.

All of the RTOSs investigated that provide support for multiple partitions use the MMU of the processor and its underlying architecture to implement the memory protection mechanism.

The common requirement is to have a system configuration table that describes the memory requirements and to use software to generate the address decoding structures into static locations. The address decoding structures are used to map virtual addresses to physical addresses, to provide control over access, and for modification over sets of virtual address spaces. Once setup, the data structures should not be modified. The setup and modification will only be performed by the MOS.

Some operating systems use the configuration tables to allocate memory for partitions and map the memory-addressing tables directly. Therefore, the presence and memory requirements for the partitions are exposed in the system configuration tables. There is a direct correspondence between information in the configuration tables and the scheduling tables. Memory-addressing tables are derived from this information.

Other operating systems use the configuration tables to derive other internal data structures dynamically at system initialization time and use this to perform memory layout and addressing table initialization.

The memory protection mechanisms should be fixed and unchanged after system initialization and before the system's application programs begin executing; otherwise, the protection of the system's memory and each partition's memory cannot be guaranteed.

The software used to transform the configuration data to a set of address translation tables must be included in the design assurance evidence for the MOS by the system integrator. Since the integrity of these tables is crucial to the safe operation of the system, all partitions, and the MOS itself, additional checks should be performed. These could be in the form of a memory verification tool that checks the page tables against the configuration information and internal data structures. These checks should be independent of the code used to generate the address translation tables and would verify the properties of the virtual address spaces against the physical layout, for example, overlapping memory and read/write attribute settings.

3.2.2 Memory Management Within Partition Space.

Because the RTOS architectures are diverse, the approaches to memory management within partitions vary. A system application requires space for several different uses of memory, as follows:

- Code. Code is statically allocated. Its size is fixed and known when the linking process is complete. The linking process could be partial and incremental to permit subsystems to be integrated, if the linker supports it. The linker can also perform subprogram elimination to remove functions that are not called (typically provided for Ada language linking processes). Whatever the linking mechanism, the code can be loaded and the protection mechanism setup.

- Constants. The constants used by the application may be gathered together by the linker into memory area, which is then loaded and protected for use by the application.

- Static Data. The linker may calculate the size of global data areas for the system application and reference specific locations based on calculated offsets for this memory. The configuration table may be used to specify the location in memory and additional attributes for these data areas. The data areas will be protected using the usual memory protection mechanisms. Additional attributes may be added to give these locations

special properties. For example, data should not be held in cache memory but always stored in memory to maintain coherency with memory-based I/O devices.

- Main Stack. The system application code runs in the context of a stack. Before any partition's processes are started, the system application's code runs using the application main stack. Either the maximum size of the stack is specified by the application developed or a default size is used. This stack area may be split into two stacks: a primary stack growing in one direction and a secondary stack growing in the other. Stack overflow occurs when the two stacks collide. Secondary stacks are typically used to allocate memory for dynamically sized objects whose lifetime can be managed in a last-in first-out (LIFO) order. Typical uses for secondary stacks are declaration of dynamically sized arrays (arrays whose size is known at execution time) and intermediate objects that exist inside the stack frame that is being exited, and so on.

  Some implementations avoid the use of secondary stacks and use the main stack instead. This is achieved by the compiler generating special code that delays collapsing stack frames and indirectly referencing dynamic data. Other system implementations use the heap mechanism to store and manage these dynamically sized objects. The use of the heap must be treated with care to avoid memory fragmentation.

  It is possible for a system application (or a partition) to run out of stack space as a stack is being used. This could occur because the stack size estimate was too small, dynamically sized objects were too big or the calling depth was too large (typically caused through direct or indirect recursion or re-entrancy). Some system implementations protect against this by performing stack checks on every function call. Other implementations place a protected page at the end of the stack, such that when the subprogram accesses a memory location beyond the stack, a memory access error is raised. This protected page mechanism precludes the use of a secondary stack from the same memory area as the primary stack. Unless each entry in the called stack frame is initialized, this mechanism is not foolproof. Some language compilers provide data with initial values but others need not. A situation may arise where the stack frame is larger than the protected page and writing to the end of the data frame may write to an area beyond the stack.

  The stack size analysis is a known design assurance problem and should be addressed to ensure that the hazard does not affect the integrity of the application.

- Heap. Most programming languages used in avionics software offer the ability to allocate memory for system application use from some pool of memory. The pool of memory may be managed through language primitives (e.g., Ada), or they may be explicitly allocated (e.g., C). The pool of memory is commonly known as the heap. The application developer specifies the size of the application-level heap. The size and location should be fixed and protected before the application starts running. The size would either be specified in the configuration table or taken from the data area set aside for each partition at initialization time. The specific method by which this is accomplished is not specified in ARINC 653.

Use of heap memory may present a design assurance difficulty unless certain precautions are taken. Allocation of memory from the heap may continue until there is no more memory available in the heap. As memory is dynamically claimed from the heap, it is important to ensure that the allocation attempts do not exceed available heap space. This could be verified by analysis of the system application. Some operating systems provide mechanisms that raise an error if an allocation is attempted after a flag is set signaling that no more heap allocations are permitted. This mechanism can be used to verify, during testing, that heap space is not claimed after some agreed initialization phase is completed and could possibly be used to capture an improper allocation during system operation.

Some operating systems permit allocation and deallocation of memory from the heap. If the deallocations occur in LIFO order, problems are averted as the heap behaves like a secondary stack. If deallocations do not occur in LIFO order, a level of uncertainty is introduced. The deallocations, which occur in the middle of a chain of allocated objects, create holes or fragments of deallocated memory. Subsequent allocations will attempt to reuse these holes. The holes are typically chained together to enable the allocation algorithm to find an appropriate hole to reuse for a new allocation. The search to find the appropriate hole is nondeterministic, and many algorithms have been proposed to help find the appropriate fragment of memory to use. First-fit finds the first available hole, and best fit finds the hole that leaves the least unused space. A buddy algorithm coalesces adjacent holes into bigger ones. Garbage collection algorithms attempt to move memory areas at run time to reduce the fragmentation, typically running as a background process. For safety-critical applications, deallocation of individual memory areas should not be permitted, other than at points in the application where timing is not critical (e.g., initialization or system reconfiguration).

It is unlikely that any system implementation will provide process-specific heap space. The most likely implementation is to have a single heap for each partition.

- Process Stack. As each process is created, it is provided with its own stack so that it may run as an independent thread. Different RTOSs implement this in different ways. The ARINC 653 CREATE_PROCESS procedure accepts a stack size as an attribute of the process to be created. The space for this process stack could be allocated from the heap space provided for the partition. Some RTOSs allow the user to specify the size of the partition's stack in the configuration table, whereas the process stack space is allocated by the RTOS at system initialization time. While the two approaches are identical once the partitions are created, they are incompatible from the ARINC 653 API unless the stack size requested through the call of the creation procedure is simply ignored. The size value used must be carefully designed and verified during the development process.

3.2.3 Control and Data Coupling Considerations.

A system application composed of many processes and multiple processes will be linked with the appropriate support routines visible to the application process in its partition. Depending upon the organization, the POS may be linked with the application processes, or a set of

interfacing routines will be linked that issue a system trap instruction, and control is passed to the POS in the Supervisor mode.

The system application linking process introduces a new set of problems for addressing control and data coupling objective no. 8 of table A-7 in DO-178B. This objective is a measure of the integrity of integration. Integration is complicated by the separation between the User mode and the Supervisor mode. There is only indirect control coupling between application-linked code and the MOS. The coupling is implemented by a system trap with additional data, which is interpreted and possibly validated. The data specifies the actual code that must be executed at the Supervisor mode. If the POS and MOS are both placed in the Supervisor mode, then control coupling will be substantially different compared to an implementation where only the MOS is at the Supervisor mode.

The partition's process will have certain characteristics that are provided to the system integrator: the name, code position, data-area position, data-area size, and so on. Unless the implementation requires it, neither the internal structure nor the names and sizes of processes within the partition need to be known by the system integrator. During the discussions of the ARINC 653 committee, two approaches became evident: (1) an approach where the process names and characteristics are exposed and (2) an approach where process names and characteristics are held private. As a compromise, the process attributes have been specified, but their use is optional. This implies that partition processes' developers may need to be open about their internal code structure, even if the system integrator is a competitor, which may raise issues if the system is being used in a secure environment.

3.2.4 Partitioned Time Protection Mechanisms.

The fundamental concept of a partitioned multiple-application system is that the partition's processes cannot influence each other's behavior, including timing, or can only influence other processes in a well-defined, verified, and controlled manner (e.g., a process in one partition may set a condition or flag for a process in another partition that effects its execution). ARINC 653 partitions run in a strict round-robin time frame with durations and periods specified in the configuration table. All implementations will enforce this time frame using a timed interrupt mechanism. The intended start time of the next partition can be accurately determined. The timed interrupt mechanism may be implemented in two ways: some processors have a high-resolution decrementing counter on the chip itself; others rely on an external timer device that generates an interrupt recognized by the processor.

Two different mechanisms are used to manage the advance of time: ticking clocks or alarm clocks. Each mechanism is discussed below.

A ticking clock is programmed to generate an interrupt when a fixed time duration expires, the tick. Depending upon system implementation, ticks could be coarse, (e.g., 20 milliseconds) or fine (e.g., 1 millisecond). As each tick occurs, some software-managed clock routine advances the system time, which may cause the scheduler to take action.

Alarm clocks require less intervention, but greater setup than ticking clocks. The clock may be either a decrement counter clock or an external alarm clock that is programmed to send a time interrupt when an appropriate time has expired. The appropriate time is calculated by inspecting the scheduling queues, determining which time event is expected next, and after what time interval it should occur. The alarm clock is programmed to respond after this interval. If a scheduling event occurs when a time event is required in a shorter period of time, the alarm clock is reprogrammed and the event that has been distracted is saved on a queue.

Fundamentally, the ticking and alarm clock mechanisms are equivalent but the following should be noted. The precision of the ticking clock should be fine enough to allow time event scheduling with the precision required by the applications. The size of the data type that implements the measure of time should be large enough to hold the time without the time rolling over. Some RTOSs have mechanisms that detect the roll over and adjust all of the values in the delay queue. This adjustment causes a problem with timing jitter because the rollover takes time from the application in which it occurs. It is difficult to consider this when programming. To avoid this problem, RTOSs can make the tick less fine or make the size of the data type larger.

Depending on the RTOS architecture, the tick delivery may require special treatment. The tick will always arrive in the MOS. If the POS and MOS are programmed to be the same kernel, time management is straightforward. If the POS is linked with the application and runs in the partition space, a tick delivery mechanism must be developed to deliver the tick from the System mode to the User mode.

3.2.4.1 Interrupt Handling.

Interrupts are asynchronous events; their arrival is unpredictable. RTOS design should balance interrupt latency against the ease of implementation and efficiency.

Because interrupts are asynchronous, it is important to protect the update to complex structures within the RTOS itself. An update to certain data structures must be completed in an indivisible operation. One approach to accomplish this is to disable interrupts at the start of the update, and to re-enable them when the update is complete. These updating operations should be kept short, because the longer they are, the longer the jitter is introduced by the lock/unlock mechanism.

Some RTOSs introduce a special queue to hold process-scheduling operations that have been delayed until the data update is complete. This mechanism ensures that the interrupt response is maximized by shortening the interrupt blocking times.

3.2.4.2 Timing Complications Due to Scheduler Design.

Scheduling algorithms typically insert a task into the ready queue based on priority. The algorithm for insertion could be a simple linear search of the queue. This makes the scheduling algorithm less predictable. The insertion depends on the priority of the process being inserted and its placement in the queue. Some RTOSs reduce this unpredictability by organizing the ready queue in blocks based on priority ranges. An initial search will search for the block to be searched, followed by a search within the block. This may reduce the average search of the

ready queue and may make the worst-case search much smaller. Other RTOSs use a more sophisticated search algorithm by using special hardware instructions so that the search is performed in a fixed time irrespective of the position of the process in the queue. The algorithm used should be described in terms that help the application developer.

3.3 SCHEDULING SCHEMES IN PARTITIONED MULTIPLE-APPLICATION SYSTEMS.

Process scheduling is used to provide some control over independent threads of execution (processes), which share the processor execution cycles.

3.3.1 Round Robin for Partitions.

The ARINC 653 model requires that partitions be executed using a round-robin sequence. Essentially, this establishes a time frame called the major time frame. Time slots are defined within the major time frame. The time slots are of fixed durations, but they need not be the same. The configuration table describes which partitions will execute in which time slot within a frame. A partition may be allocated to more than one slot within a frame.

Execution of each partition follows in sequence, starting at the beginning of a frame. When the last partition in the frame is executed, the sequence is repeated. The time slots are strictly enforced by the MOS. A clock device is used to ensure the timely switching between partitions.

3.3.2 Processes Inside Partitions May Also Be Periodic.

ARINC 653 provides an API through which users may create processes within a partition. During their creation, certain processes could be specified to be periodic. A time period may be given that specifies an interval, at the end of which the process will restart. The expectation is that the process will give up the processor after a shorter period than the execution interval. Many such tasks could coexist within a partition, provided they can share the available processor time while their partition is running. For applications that require executing some fixed algorithms in a periodic way, this mechanism could be appropriate.

A duration replenish operation is provided in ARINC 653 that permits applications to adjust their allocated time duration for specific time frames. This approach complicates timing analysis but may be an appropriate solution for high-priority processes.

3.3.3 Pre-Emptive for Processes.

In ARINC 653, processes are pre-emptive. In periodic systems, it may be possible for processes to be cooperative, and a process will execute until it voluntarily gives up the processor. In ARINC 653, processes are pre-emptive, even periodic ones. If a periodic process does not give up the processor when its period ends, a timer interrupts it and the scheduler takes control. In addition to periodic processes, ARINC 653 permits the creation of aperiodic processes, which are selected for execution by the scheduler based on certain internal or external events. The events may be synchronization primitives; for example, message buffers, semaphores, or simple timeout mechanisms.

3.3.4 Timeouts.

Certain process operations may be unable to complete immediately because they are blocked while they await response from another process. The blocking time may be limited by a timeout value, such that the operation that was blocked is released when the time limit is reached. The release mechanism is time triggered; hence, the requirement for a pre-emptive scheduler.

The algorithms that perform scheduling in an RTOS are designed to meet various design objectives. The RTOS designers make their design choices for many reasons. The RTOS scheduling implementations are varied, and appropriate design assurance data should be available.

In the ARINC 653 specification, two scheduling mechanisms are suggested. The MOS provides scheduling for partitions, and the POS provides scheduling for processes within a partition. A natural mapping would be to put the MOS in the Supervisor mode and the POS in the User mode. This means that the processes may be linked with the POS, making the calls to each of the POS functions efficient. Calls to the MOS are translated to system call instructions, where the operand provides access to parameters that provide the identity of the actual call being made and additional data as required. There is an overhead when using the system call mechanism: registers and other context information must be saved and restored, modes switched, and the parameters verified.

An alternative to mapping would be to put the POS and the MOS into the Supervisor mode and to treat all calls as system calls. This makes the implementation easier but places a greater overhead on all process operations and scheduling calls for both partition scheduling and communicating.

It is important to obtain timing information for these scheduling operations not only for the purpose of meeting DO-178B objectives for the RTOSs, but also to ensure that the application developers will be able to estimate how long their processes will take and to ensure that they can guarantee schedulability.

The speed of the processor could be many times faster than the speed of some I/O devices. A process that sends data to a slow device may be blocked while the device accepts the data being sent. Imagine a User-mode process that is writing a message to a buffer, which is forwarded to the I/O device. As soon as the buffer is full, the process is blocked. In a nonpartitioned system, the scheduler would be invoked to select the process with the next highest priority that is ready to run. When the I/O device empties the buffer, an interrupt would be sent to the scheduler to unblock the blocked process.

In this system, which combines the schedulers in the Supervisor mode area, the scheduler can easily find the blocked process, change its status, and reschedule. In a system where the schedulers are separate, an interrupt mechanism must be used. However, interrupts are not permitted in the POS layer. To overcome this, the interrupts must be simulated and some event mechanism must be used to force the scheduler to perform its work.

Some design alternatives may optimize this by providing device drivers with processes that run in the MOS layer. A User process would still send data to a buffer in the MOS layer, and a hidden device process would provide the decoupling required between the device and the buffer management process. In this type of scenario, it is important that the process, which is performing I/O on behalf of some User-mode process, only does so in the partition that owns the User process.

4. PARTITION-SUPPORTING RTOS CONSIDERATIONS.

This section builds upon the RTOS and partitioned multiple-application system architectures presented in section 3. Specific considerations for using partition-supporting RTOSs in IMA systems are considered, such as BSP, cache jitter, background partition, data flow between partitions, worst-case execution time analysis, scheduling within partitions, integration, and RTOS libraries.

4.1 BOARD SUPPORT PACKAGE CONSIDERATIONS.

The BSP is software that interfaces the RTOS to the target computer. The BSP is typically the first software component executed (the RTOS is entered and makes a call of the BSP functions). The BSP initializes the processor, initializes memory, performs various memory checks, initializes devices, and so on. Once the initializations are complete, the BSP returns control back to the RTOS, but its functions are still available to perform hardware-related functions such as low-level cache manipulations. Clearly, such low-level functionality can only be performed if the BSP has access to the privileged instructions. This means that the BSP must execute at the System mode.

The BSP will also interface to target-specific components such as interrupts, clocks, and timers. In typical RTOS implementations, access to devices is also programmed in the BSP. Device-independent code may be implemented in RTOS device libraries, but the low-level interface routines are typically implemented in the BSP.

In multiple-partition systems, the location of device driver code is more complex. A device driver may simply be placed in a partition. The I/O may be performed through a memory buffer that is mapped at an address location known to the device itself. If the device code is entirely in the partition space, then only polled I/O mechanisms are permitted. This means the I/O device is only available to a single partition, and its I/O operations will only take place while that partition's process is running.

An alternative approach is to place part of the device driver's processes into a single partition and part of the device driver's processes into the MOS. Partition's processes may send and receive data using the interpartition communication mechanisms provided by ARINC 653. The single I/O partition may be scheduled with many short durations to reduce I/O operation latency. The I/O drivers could be interrupt-driven to improve the responsiveness and increase the coupling between the processor and the device.

It is also possible to place the I/O device driver's processes entirely in the MOS. This may be the most efficient approach, but it must be handled carefully. If the device driver is not adequately decoupled from the scheduling of the MOS, then a single device could affect the time frame of a partition that has no interest in the device being used. This violates the principles of partitioning. One approach is to introduce jitter margins that will be taken into account when programming the final application. The other approach is to introduce hidden threads that perform the I/O operations, which can be pre-empted when scheduling is required on a partition switch or some other higher-priority operation.

From a design assurance point of view, an RTOS must be evaluated with its BSP as part of the project under certification. By putting more of the code into a partition and less into the BSP that resides in the MOS, it becomes easier and less expensive to adapt the system to a new hardware configuration that contains a new device driver without submitting the RTOS for reapproval.

A further complication to I/O operation is that the memory seen by a device and the device itself describes the state of the device at a particular point in time. If a device is controlled by one partition only, then the partition can be restarted because the state of the device can be determined. If an I/O device is used by several partitions, then the partitions cannot be restarted because its state relative to those partitions cannot be determined.

4.2 CACHE JITTER.

Cache memory is a global resource that is shared between partitions. The information in cache may contain page address values, data values, and code. When a reference to a particular value is encountered, an automatic search is performed in the cache memory. If the value is found, it is used directly (cache hit); if the value is not found, it must be loaded (cache miss). The load is much slower, as it requires an off-chip memory access cycle. Once loaded, the value may be reused for as long as it remains in cache memory. The cache is very fast but expensive (per bit), therefore, it is much smaller than typical off-chip memories. See section 2.2.6 for details.

Assume a simple scenario of four partitions (P1, P2, P3, and P4) where these partitions have a duration of 25 msec and a period of 100 msec. According to the rules for partitioning, it must be impossible for any partition to influence any other or to only affect other partitions in well-defined and controlled ways. During integration testing, it is determined that all partitions complete their processing in their allotted time slots. This may have been accomplished because partitions P1, P2, and P3 perform some very intensive computations that use little data and very little code during the computations. Code can consist of small, highly computational loops.

Under these conditions, three partitions may use very little of the cache. Partition P4 will have much of the cache available to itself, and it will have very few cache misses in each iterative period. Under normal conditions, P4 will complete its work within its allotted period.

As an example, assume P3 suddenly changes its mode of operation, such that it references a large amount of data and executes a lot of code just once rather than referencing a small amount of code iteratively. In this case, P3 will cause P4 to have many cache misses for both code and

data. This may result in P4 growing by a factor large enough to cause it to miss its deadline. Clearly, this violates the principles of partitioning, because one partition has caused another to fail, which is unacceptable.

The problem may be lessened by selecting different caching modes. In copy-back mode, a data value is held in the cache until space is required for a new value. Before the cache memory location obtains its new value, the old value is written to the target memory location. The memory store and memory load are still performed every time a data value is updated, but several updates in memory may be performed without extraneous reads and writes. Absence of cache misses reduces the reads and copying back to memory.

A copy-through cache policy forces the value in cache to be copied back to memory each time a value is updated. As long as a value is in cache, it will be reused, but as soon as it is changed, it is written back to memory. This policy reduces the read from memory, but it does not decrease the writes.

It may appear that the copy-through policy is always more efficient and should be used exclusively, but this may or may not be the case, depending on the distribution of the data accesses.

A data value read results in the placement of the value in the referenced memory location as well as the adjacent locations. Depending on cache line length, this could be a 32-byte memory fetch and a corresponding 32-byte write. If a copy-through policy is used, then only one memory location is written. If memory reference density is high, then the copy-back policy is more efficient; otherwise, the write-through may be more efficient, depending upon the program.

Data reads are not affected. Because code is not copied back, it is unaffected by the copy-back mode.

There are several solutions to make the cache behavior more predictable:

- Switch the cache off. While this may seem attractive because it makes each of the partition execution profiles deterministic, it places a large performance penalty indiscriminately over all the partitions.

  Most programs assume the availability of cache memory and expect the performance improvement that they are given.

- Specify a huge jitter margin. When designing the partitioned system, specify that each partition switch will include a margin that is equivalent to a cache miss on every value in the cache. This overhead is subtracted from each partition's duration. Each process in a partition must be verified independently under these timing conditions.

  The disadvantage to this solution is that each partition is subject to the same fixed margin. A partition with a very short duration looses a large percentage of its permissible processing capability, especially if the cache is large.

- Selective flushing. For those system applications that require very deterministic performance, the cache could be flushed during the partition switch such that the incoming partition has a clear cache memory at the start of its duration. Flushing means copying all the cache values only present in the cache back to main memory (i.e., they have been updated, and copy-back mode is used).

  This places the overhead at the start of the partition rather than it being distributed throughout. The time taken to perform this operation is not fixed, as it depends on the number of values that must be written to memory.

- Selective Invalidate. For system applications executed in a write-through mode for which deterministic execution is required, the cache may be invalidated during a partition switch. A cache invalidate is very fast. A single instruction makes the cache appear empty. Subsequent data reads are stored in the cache and reused as normal.

The start of a partition can be precisely predicted. The timer interrupt to start the partition switch should be very accurate, and the time to switch context (save and restore all of the registers) will be constant. Additional overhead to select the next partition should also be small, and its maximum well specified, especially as the partition scheduling is based on a simple round-robin algorithm.

The questions for design assurance include:

- Does the behavior of the cache affect time allocated to a partition?

- Can the worst-case partition execution time be tested, by flushing the cache in a preceding partition window?

- Is there a mechanism provided to control the cache behavior at partition switches?

- Does the system integrator control the cache behavior through the configuration tables?

The loss of processing throughput due to cache misses becomes more important if it affects short time events. A partition may have processes that perform calculations and which synchronize with other processes. These synchronization events may be issued with time outs so that if a process is blocked, its blocking time is limited. In the presence of delays caused by cache misses, the calculation times, and thus synchronization responses, may be affected. Cache misses can affect the behavior of processes within a partition, even though the effect can be bounded when analyzed over a partition's duration.

Design assurance concerns must include not only the cache-induced jitter on partition start and duration, but the effect on internal timing events as well.

4.3 BACKGROUND PARTITION TIME.

A fully configured system consists of a number of applications. Each of these applications runs on the target computer for a prescribed duration in a fixed, repetitive time slot as specified in the system configuration table. A specific application may have a number of processes that run periodically or that run when triggered by some event. The event could be a message, a timeout, a semaphore state change, and so on. The application could reach a state where all the periodic processes are blocked for some reason. They could be waiting for an event, or for some time to expire. At this point, the application is in an idle state. Typically, the scheduler cycles through an algorithm that continually checks for a change in status. The computer is not performing any useful calculations but is continually monitoring exception status settings, which trigger when a time event or an external event arrives.

An application that is cycling in idle state may be perceived as wasting valuable computing resources. The implementers of some multiple-partition systems view this as a processing resource that could be used to perform useful background work. A system has been proposed where processes are identified during their creation as PRIORITY processes. These PRIORITY processes may be created in any partition, and they are given a system wide priority. PRIORITY processes do not run until a partition is about to enter an idle state. At this point, the scheduler will select a PRIORITY process based on its priority, irrespective of which partition it is located. If the highest priority process is in another partition, then a partition swap occurs. The key concept is that the PRIORITY processes always run in the partition in which they are created (i.e., they can only access data available in the partition), and PRIORITY processes only run while the non-PRIORITY processes continue to be blocked in the partition that is scheduled to run.

The intention of this approach is to use processing power for noncritical processes that would not affect safety if they were not run at all.

The time taken to switch to the background PRIORITY processes and the time taken by the background PRIORITY processes should not be a design assurance concern.

The time taken to switch from a background PRIORITY process back to a partition in which a process has woken up is a concern. The time is lost from the partition time. If the processes rely on responsiveness during their partition time, then determinism could be an issue if responsiveness is compromised by background PRIORITY processes. One way to avoid a partition switch is to have a background process that absorbs CPU time and does not allow the partition to become idle.

4.4 DATA FLOW BETWEEN PARTITIONS AND PROTECTION MECHANISMS.

Information may be sent between partitions in a number of ways. The ARINC 653 specification offers a message-passing mechanism where a partition transfers a message using send or receive functions using specified channels between nominated partitions. The communication protocol verifies that the partitions and channel connections are defined in the system configuration table. If specific permission is not granted, then the communication must be failed with appropriate

errors logged to the Health Monitoring system. A blackboard mechanism is specified in ARINC 653. Using a blackboard object, one partition may write to the blackboard and several other partitions can only read from the blackboard. This mechanism serves as a broadcast message system, where a message may be updated at one rate and read at different rates by the reading processes. There are many ways to implement these data-passing mechanisms. Four such approaches are described in this section.

The buffers used to write and read the message must not be visible to any single partition, as that would compromise the space partitioning. Various design options are used to implement this requirement.

4.4.1 Direct Copy by Kernel.

In the direct copy by kernel approach, the kernel acts on behalf of the writer. As shown in figure 2, the message is put into the memory assigned to the writing partition at the time of the write message call. When the message is read by the receiving partition, then the kernel transfers it from the write buffer area to the read buffer area. The actual transfer is performed during the period that the receiving partition is running. Because the kernel can see all the memory, the message transfer is straightforward. This technique is used mostly by kernels that combine the MOS and POS in the kernel.

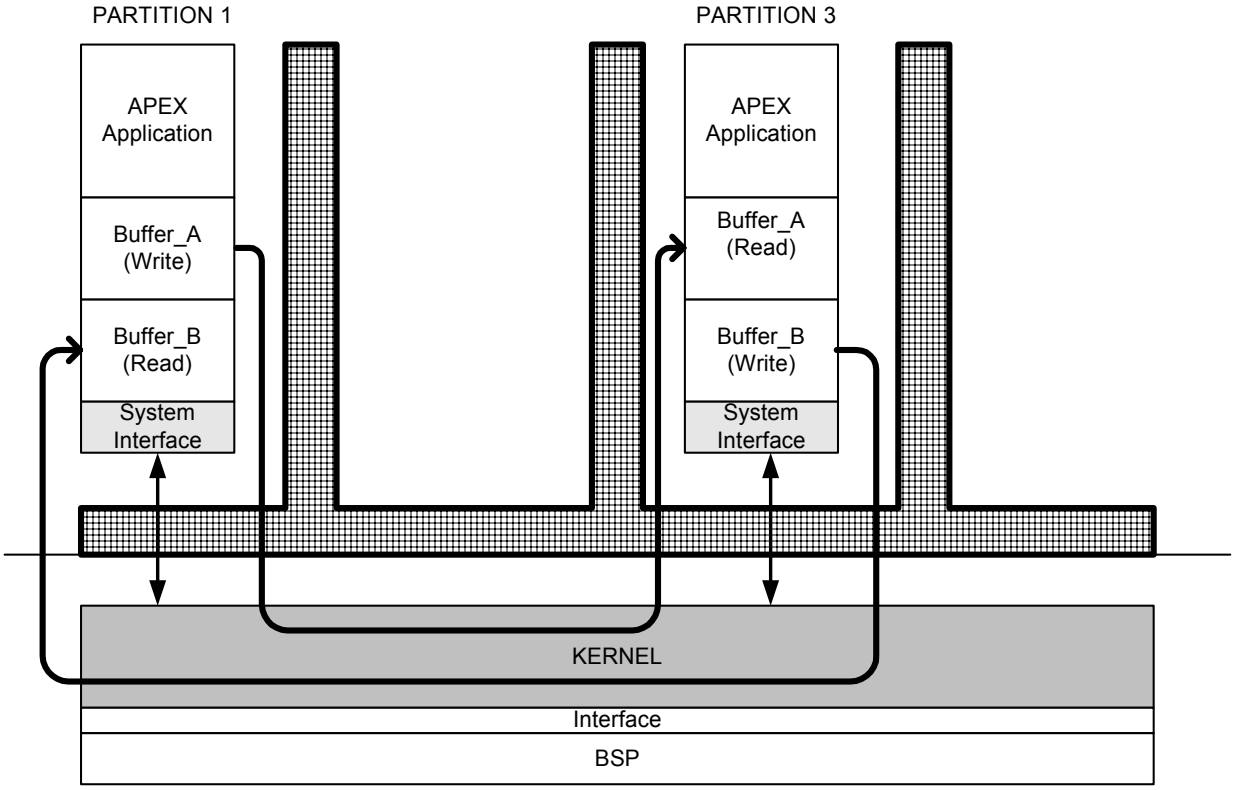

FIGURE 2. KERNEL COPIES FROM PARTITION TO PARTITION SPACE DIRECTLY

The semantics of message transfer using this approach are complex. The messages in the buffer are based on FIFO order. Message writes must be blocked if the buffer is full, and message reads must be blocked if the buffer is empty. The order of unblocking these blocked partitions may be FIFO- or PRIORITY-based. The blocking may be limited by timeouts if requested at the point of call. The code to implement the semantics over the partition boundaries may be complex and difficult to verify and show compliance with the DO-178B objectives.

4.4.2 Indirect Copy by Kernel.

The indirect copy by kernel implementation is preferred when the kernel implements the MOS, and the partition contains the POS. Figure 3 illustrates the indirect copy by kernel approach. The message is transferred from the partition to a buffer in the kernel, and then transferred from this intermediate buffer to the destination buffer. The transfer may be performed by kernel code or by special internal processes created to manage the internal buffers. The advantage of this approach is that buffer management behavior is greatly simplified. A full intermediate buffer results in a buffer process being blocked until some data is removed. The scheduler performs process scheduling using the standard process mechanisms. This means that the specific semantics of buffer management are implemented by the intermediate buffer processes. The design assurance of this implementation may be easier because it builds the more complex semantics using process-scheduling mechanisms.

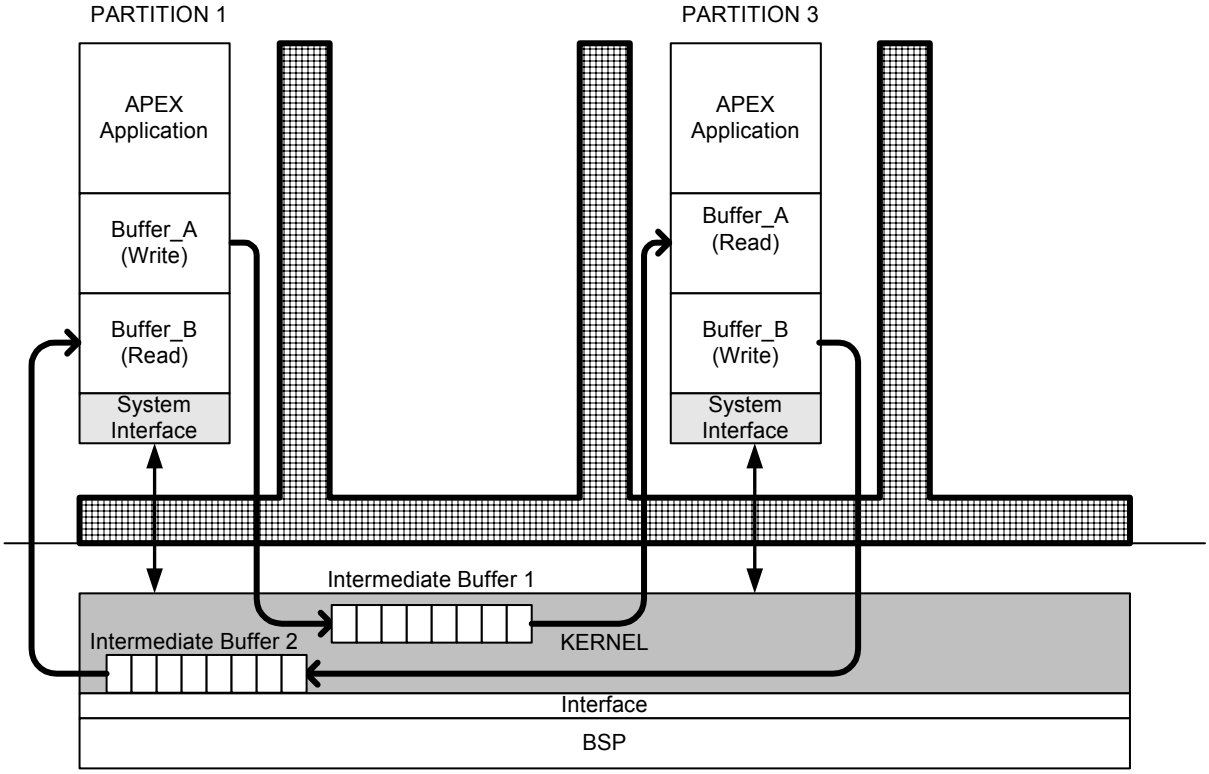

FIGURE 3. BUFFER COPIES VIA KERNEL BUFFER

4.4.3 Zero-Copy Synchronous.

In the zero-copy synchronous approach, the buffers are placed in an area of memory that is remapped when a partition is switched. In the example shown in figure 4, when Partition 1 is running, a read/write area is made addressable to contain Buffer_A, and a read-only area is mapped to contain Buffer_B. Message send and receive operations are able to read and write to the appropriate buffers provided that the correct operations are performed. Clearly, sending a message to Buffer_B would cause an error, and the kernel would respond with the appropriate error response. When Partition 3 is switched in, the page protections are reversed for the common areas. This approach has the advantage that large messages do not need to be physically copied.

The memory mapping is controlled through information in the system configuration tables, and implements the interpartition communication protocols permitted by the system integrator. The risk is dynamically modifying partitioned memory accesses and requires careful scrutiny.

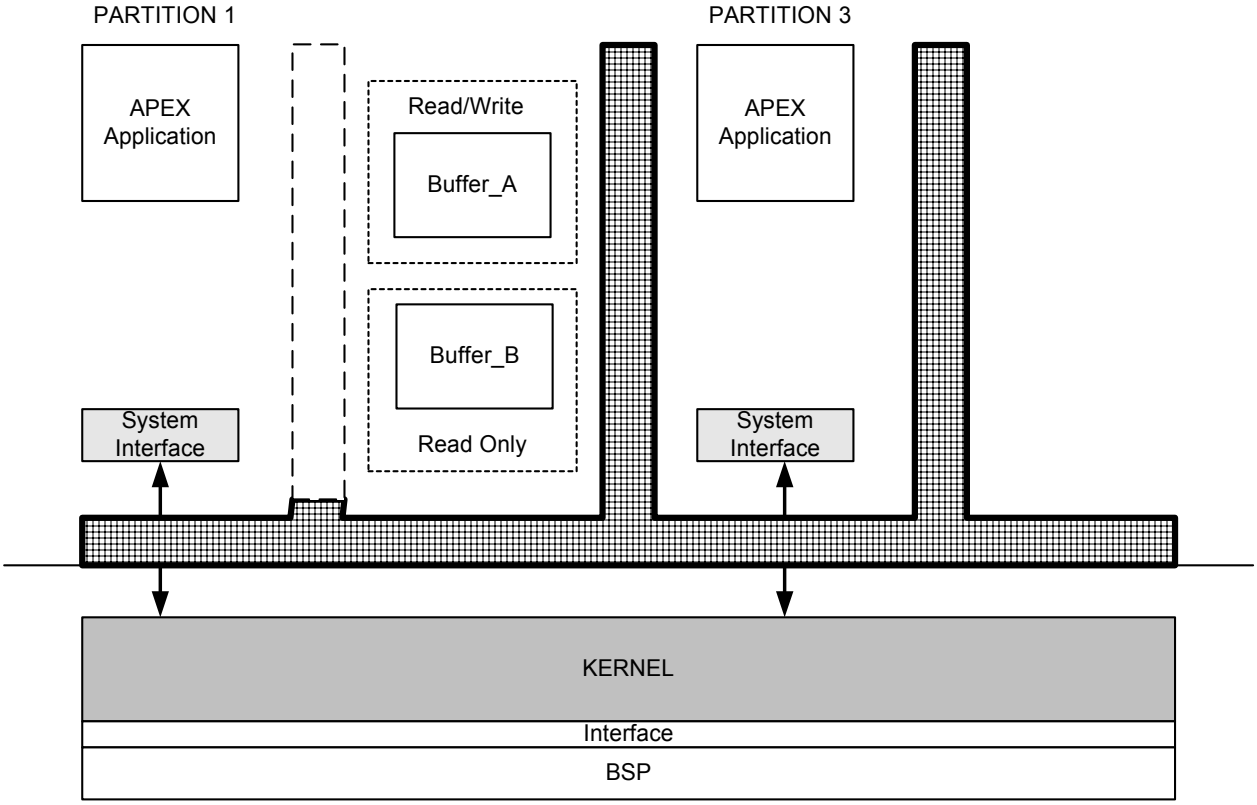

FIGURE 4. ACCESS THROUGH MANIPULATION OF PROTECTION MECHANISM

4.4.4 Zero-Copy Asynchronous.

The zero-copy asynchronous mechanism may be demand-based rather than based on a partition switch. The approach here is that the protection mechanism is only used when required on message read. Using the example in figure 4, a message may be sent by Partition 1 to Buffer_A.

A partition switch to Partition 3 would not result in memory being remapped for the buffer area. If code in Partition 3 attempted to read the message from Buffer_A, then the hardware would raise a memory access exception. This exception is intercepted by the kernel, which checks the system configuration table to see if the memory access should be permitted. If the access were permitted, then the memory would be remapped so that the read from the buffer could continue.

The advantage of this approach is that only the memory that is actually referenced during a partition's execution needs to be mapped. The mapping is only performed on demand, using the configuration tables to check.

The disadvantage of this approach is that the mechanism used to verify memory integrity is being used to actually implement the control of the access itself. This makes the MMU mechanism both the implementation of the access mechanism and the mechanism to enforce the memory separation, dynamically. The MMU should be used for only one of these.

4.4.5 Partitioning Considerations for Nonpartitioned Developed Code.

A multiple-partition programming model implies that applications are run in isolation from each other using a defined API. The API isolates the application processes from the underlying processor, except for the communication channels permitted through the configuration table. This API has also been proposed for nonpartitioned operating systems as a way of developing software for a federated system and reusing the same software in a partitioned system. This approach is perfectly valid, but it may conceal a portability problem that must be addressed.

An application developed for a federated system is likely to be linked and run at the Supervisor mode of the target processor. The application may use privileged instructions, which are accessible to the application's processes. In a partitioned environment, the instruction causes an access violation exception. Normally these instructions are used infrequently; they are used from assembly language code and may be in paths controlled by conditions. This presents the possibility that one of these instructions may be present, yet not be detected through testing.

Once the code is moved to a User-mode partition in a multiple-partition architecture, there is a risk that the privileged instructions will be encountered, causing the system application to fail. Note that this failure would not occur in a nonpartitioned environment.

To reduce the risk of this happening, two strategies are possible: (1) analysis of the executable code to ensure that these instructions are not present and (2) analysis of the object code to ensure that these instructions are not present. Coverage analysis will find all occurrences of this code if it is performed at the machine code level.

A design assurance strategy should be developed to address this potential hazard and ensure that a means of mitigating it is provided.

4.5 WORST-CASE EXECUTION TIME.

The software aspects of certification of systems with high criticalities require a proper demonstration of the WCET for those systems and must be deterministic. Proper calculation of WCET is rather hotly debated, particularly for systems using complex architectures such as the PowerPC.

Several approaches to the calculation of WCET can be taken. One approach is to model the microprocessor and its associated timing behavior. Once the model is complete, it can be used to predict the WCET. The debatable part of this approach is the completeness of the model. To precisely model the timing behavior of a complex microprocessor, such as the PowerPC, can be a tremendous undertaking, and validating that model can be very difficult. Features such as pipelining, caching, and out-of-order execution models represent some of the more difficult WCET prediction attributes. A simpler model can be used, but the model's accuracy must be determined, and any error in the calculation should be known [9]. Zhang reports that estimates based on models that do not consider pipes and caches account for 17% to 40% over estimation times. Part of the problem with a precise model is the significant technical challenge of WCET measurement or calculation in a realistic environment using modern microprocessor-based systems that contain large caches and sophisticated pipelining, branching, and other optimization techniques.

A standard industry approach to proper WCET calculations does not exist. Any system that requires determinism must specify how WCET is predicted and provide a basis for validating that prediction. One solution is to combine the calculation and measurement of various software system components, to arrive at a statistically acceptable determination of WCET. Understanding realistic worst-case timing for a set of code instructions is difficult. A strict code-only-view of estimating the timing can be nonrealistic because it can create scenarios that could not possibly happen in the physical environment. Therefore, application environment considerations provide a more realistic timing basis. The approach is to run the predicted application scenario on the target system, and obtain nonintrusive measurements to validate the estimates. In this approach, the deviation range between the calculated and measured values becomes the tolerance band around the predicted time.

COTS processors are designed for the average case performance, and therefore, WCET calculations tend to be difficult. The validity of the models used in any WCET analysis is necessary to verify its pedigree of prediction.

4.6 SCHEDULING WITHIN PARTITIONS.

Many types of scheduling schemes can exist within a partition. This section briefly describes several available schemes [10]:

- Cyclic Executive. This technique is still in widespread use and simply cycles through a set of processes whose execution order has been predetermined.

- Round Robin. This is a simple scheduling algorithm designed for time-sharing systems. A minor time slice is defined by the system, and all processes are kept in a circular queue. The CPU scheduler goes around this queue, allocating the CPU to each process for a time slice interval. New processes are added to the tail of the queue. The CPU scheduler selects the first process from the queue, sets a timer to interrupt after one time slice, and then dispatches the process. If the process is still running at the end of the time slice, the CPU is pre-empted, and the process is added to the tail of the queue. If the process finishes before the end of the time slice, the process itself releases the CPU voluntarily. Every time a process is granted the CPU, a context switch occurs that adds overhead to the process execution time.

- Fixed Priority Pre-emptive Scheduling. Each process has a fixed priority that does not change from instance to instance. A higher-priority process pre-empts a lower-priority process. Many real-time operating systems support this scheme. An alternate to this scheme is a dynamic priority pre-emptive policy, which is not typically used in safety-critical, real-time systems.

- Rate-Monotonic Scheduling. An optimal fixed priority policy where the higher the frequency of a process, the higher its priority. It can be implemented in any operating system that supports the fixed priority pre-emptive scheme. It assumes that the deadline of a periodic process is the same as its period.

- Deadline-Monotonic Scheduling. A generalization of the rate-monotonic scheduling policy, where the deadline of a process is a fixed point in time from the beginning of the period. The shorter the deadline, the higher the priority.

- Earliest Deadline First Scheduling. A dynamic priority pre-emptive policy. The deadline of a process instance is the absolute point in time by which the instance must complete. The deadline is computed when the instance is created. The scheduler selects the process that has the earliest deadline to run first. A process with an earlier deadline pre-empts a process with a later deadline. This policy minimizes the lateness of process.

- Least Slack Scheduling. This is a dynamic priority pre-emptive policy. The slack of a process instance is its absolute deadline, minus the remaining execution time for the process instance to complete. The scheduler picks the process with the shortest slack to run first. Processes with a smaller slack pre-empt processes with a larger slack.

Many variants of schedulers exist that can run within a partition. In higher-criticality systems, the scheduler should be well understood not only to confirm determinism within the partition, i.e., avoiding priority inversion, but also to coordinate interpartition activities if the system accommodates these activities. It should be noted that ARINC 653 does not support priority inversion protection. Applications written using the ARINC 653 API must be written to avoid priority inversion directly. The verification of the scheduling approaches should also be carefully considered. Users should have an approach to ensure that the scheduler provided by the RTOS is meeting their guarantees.

4.7 HEALTH MONITORING.

Health monitoring in this report is as defined in the ARINC 653 specification. Health monitoring is defined as the function of the RTOS responsible for monitoring and reporting hardware, application, and RTOS software faults and failures. The Health Monitor helps to isolate faults and to prevent failures from propagating. The specification identifies a number of error codes that must be detected and handled. The configuration tables are used to permit the system integrator to specify where error conditions should be propagated, and where error conditions should be handled. An error can be handled within a process, in a partition, or in the Health Monitor process.

Health monitor code can be used to contain the errors, to substitute alternate actions, or simply to record the errors in an error log. Errors handled by a process use the time window of the enclosing partition. Errors handled by a partition may have code executed in the MOS but use time from the partition window in which the error first occurred. Errors occurring in the MOS use time from the partitioned multiple-application system itself. As health monitoring is part of the multiple-partition implementation, its code must be verified to the same level as the RTOS.

Non-ARINC 653-compliant designs must address health monitoring as well. Noncompliant designs should be able to address how the ARINC 653-compliant health monitors are handled in the noncompliant design. In either case, the need for or the identification of additional health monitors should be addressed. The system design, hardware design, operating system design, or partitioning design may justify the need for health monitors and actions not called out by the ARINC 653 specification.

4.8 SYSTEM INTEGRATOR CONSIDERATIONS.

The final multiple-partition system must have an overall system integrator that allocates and integrates the resources of the various applications and partitions of the system. The resources used by an application must be strictly controlled, as indiscriminate resource usage could affect the behavior of another partition. Control over resource use is provided using a configuration table. The configuration table is developed by the system integrator in agreement with the specification of each application that is mapped to a partition.

The properties of a partition recorded in the configuration table may include:

- Memory (code and data).
- Time (the duration during which partitions may execute and the repetition rate, which may be expressed as a time or an execution slot number).
- Use of interpartition communication mechanism (buffers, ports).
- Use of a dedicated I/O devices.
- Health monitoring specifications (which errors will be handled at which level, and what each response should be).

Supplement 1 of ARINC 653 specifies the table format in terms of an extended markup language (XML) file with predefined tags. This is intended as one means of specifying the contents of a given configuration. This information would be translated into a form required by the actual implementation of the MOS.

This information may be in the form of an actual table that can be read using some diagnostic-style memory access mechanism, or the table may be in some internal structure that is implementation-dependent. Such information could be concentrated in a linear memory location, or it may be set up as a tree structure.

The table must be put in a memory location that is protected from access by the partitions and their processes. The table could be placed in the Supervisor memory region, or it could be placed in a special memory partition that contains only data and is not scheduled.

To provide greater flexibility, several configuration tables may be loaded and a mechanism provided that switches between the configuration tables in response to a specific event. This could be used to setup a mode switch mechanism, which can change the partition's execution profile. For example, an errant partition could be suspended while a substitute partition, running a simpler algorithm, could take over in a get-me-home mode.

4.9 REAL-TIME OPERATING SYSTEM LIBRARY CONSIDERATIONS.

The programming interface for a particular programming language may provide functionality that is specified in the language, but is actually provided by run-time code. In the C language, a number of standard library specifications permit the user to call functions to move memory, compare strings, and use mathematical functions such as mod, floor, and cos. In the Ada language, similar functionality may be provided through the language itself; for example, an object assignment where the objects are declared as arrays, or an object comparison where the objects are declared as records. In addition, predefined packages may offer mathematical functions.

In Ada, the run-time libraries are typically provided by compiler vendors and, as such, are classified as COTS software. In C, the libraries can be provided by the compiler vendors, third-party suppliers, or in the operating system.

If such run-time library code is used in an airborne system, then it must also meet the objectives of DO-178B. The following issues should also be addressed:

- Where does the code reside? It is very unlikely that it will be placed exclusively in the Supervisor area because the run-time overhead invokes the library functions from the User mode.

- Is the code shared or replicated? Certain library functions may be used at the Supervisor mode and by the application. For example, it is likely that the C language memory move functions would be used at both levels. It is possible to arrange that there be just one physical copy of the run-time libraries and map to them from the Supervisor mode to

more than one partition. This would require a special linking mechanism and all the run-time libraries would need to be present in memory; otherwise, the addition of one function could potentially require a relink of all the functions in the system. Therefore, the library may contain functions that are not used in a certain combination of loaded partitions. The code would be deactivated, providing that derived requirements existed for all functions whether they are used or not.

- Is the code able to be pre-empted and re-entrant? Can a run-time library function be pre-empted while one process is using it and the next process that becomes active also invokes the same run-time library function?

- Does the code make use of static or global data? Use of this type of data tends to make a function not re-entrant, since recursive calls to the function may overwrite previously established calculations.

- Is selective linking used? This is typically used in Ada programs. The entry point of an application is identified, and all the code references in a transitive closure are linked together so that each application has called a copy of the library functions only. While this eases the analysis of functions that are not actually used, it complicates the control coupling analysis because each link with a test program is different, as the test programs are different.

5. SAFETY IMPLICATIONS WITH RTOS WITH SECURITY-BASED FUNCTIONS.

A multiple-partition system may have a requirement to support multiple-level security. Military applications that also require compliance with the DO-178B objectives have the strongest motivation for security. It is likely that in the future, this requirement will also be imposed on civil applications. As more federated systems are placed on integrated systems, there is an opportunity to integrate information to make the overall flight more efficient. Flight management system data, integrated with global positioning system (GPS) data, integrated with Automated Dependent Surveillance-Broadcast data could ultimately yield optimized information to guide the aircraft and pilot.

If a rogue, low-criticality program that is not subject to the rigor of certification design assurance was inserted and obtained access to the shared navigation data, it could transmit a guidance trajectory to a missile. Note that there are many if's in this scenario. The security domain attempts to address these issues and ensure that information can only flow through controlled paths.

The security domain is governed by a document entitled "The Common Criteria" (CC). This document describes seven levels of security, with Level 7 being the highest. Integrated systems needing to support the CC security model would need to be certified by the security certification authorities. The operating system at the core of the integrated system would need to be certified to the highest level of partition permitted to run on the integrated system.

Similar to DO-178B, the CC approach requires a rigorous software development and verification process. A study of the mapping between DO-178B and CC show that a system certified to Level A under DO-178B would be very close to satisfying the objectives of CC at Level 5. With some additional rigor in the configuration management objectives, it could be possible to satisfy the CC objectives at Level 5 for an RTOS certified to Level A.

The main interest for the security domain is to prevent information from flowing from one partition to another, unless this flow is authorized. The phrase used in the CC documents is "the absence of covert channels," which can be viewed as a blocking data flow channels, which may compromise a system's security due to a covert infiltration.

The partitioning protection mechanisms in an integrated system have exactly the same protection of data access objectives. An RTOS in an integrated system will ensure that data cannot be read or written unless the configuration settings permit. The security domain takes this a step further and raises concerns about not only access to the data objects themselves, but of their values. A data value may be manipulated in intermediate locations that could be globally accessed, for example, registers or cache memory. A partition switch may require these locations to be sanitized (overwritten with null values) to ensure that information does not propagate as a covert channel. It may not be obvious that information can be passed through registers on a context switch, however, some RTOS's may attempt to optimize a context switch, depending on register usage. If a partition does not use floating-point operations at all, then it may be redundant to save and restore these registers. Under these conditions, it may form a covert channel that can be exploited by a partition that does not use floating-point operations.

A security-based system may require the cache to be flushed on partition switches. It could be demonstrated that cache addresses and values are mapped in pairs, and provided physical addresses do not overlap, the value is unique to a physical location. This means that on a partition switch, the values cannot be reached by a partition that does not map to the same addresses. The security domain is more conservative. It may be a requirement that the cache must be sanitized on partition switches, interpartition message buffers must be sanitized, and so on.

A possible covert channel could be the use of a global resource that is shared between partitions. The direct use of the resource could be prevented with some interlock or timeout mechanism. However, the availability of a resource itself could be used as a flag for the data communication without actually passing data as a value. Blocking a resource for a given time duration could be used as a covert channel. The use of time events has been recognized as a mechanism for establishing a covert channel. To prevent such channels, the security domain seeks to eliminate this possibility by stipulating the inclusion of an amount of nondeterminism. The safety certification domain requires determinism because time plays a crucial factor in real-time control systems. A system that addresses both concerns will require careful construction and verification.

At the highest level of security certification, the degree of rigor is severe. The appropriate security authority will perform the analysis. Within the United States, this is the National Security Agency (NSA). Formal analysis is used, requiring that the code be mapped onto

defined states and state transitions. These must be specified using a formal specification language. The formal specification is then subject to analysis using formal methods to verify the security properties. Such analysis is slow and expensive and tends to grow disproportionately with the size of the system specified. To ease the burden on the NSA, a limit of 4000 lines of code is set on the RTOS code responsible for enforcing the isolation of partitions.

6. ROBUSTNESS TESTING CASE STUDY.

6.1 REVIEW OF PHASE 3.

A previous phase of this study (Phase 3) explored the COTS RTOS domain and its safety implications. The Phase 3 report provided a detailed look into the safety and system certification issues of using a COTS RTOS in aviation applications. RTOS attributes were detailed and their safety-related properties were discussed. A stress or robustness test plan was presented as an example for the basis of an RTOS vulnerability analysis.

6.2 CASE STUDY BACKGROUND.

The stress test plan developed in Phase 3 was provided to an application executing on a multiple-partitioned RTOS on a single CPU. The applicant's RTOS was customized from a vendor's COTS RTOS to incorporate the partitioning requirements of their system. The applicant reviewed the stress test plan provided to them and added testing that was not previously identified in their test plan. Execution of the stress test plan was to demonstrate the robustness objective A6-4 of DO-178B. The goal was to determine the effectiveness of the RTOS stress test plan offered in Phase 3, and the adequacy of this particular RTOS to meet additional stress test cases.

6.3 CASE STUDY OBSERVATIONS.

Several observations were made during the case study, as summarized below:

- The applicant's customization of the COTS RTOS was extensive, in part because of the RTOS's inability to handle multiple partitioning on a single CPU and because the RTOS lacked some of the rigor suggested by DO-178B. Of particular interest in this exercise is the applicant had already included many of the stress test attributes as part of their system requirements. As such, they did not have to augment many additional stress tests since their stress testing became part of the normal functional testing, as indicated by their requirement traceability matrix.

  Since robustness was part of the requirements in this case, a separate stress test plan did not uncover many omissions, if any. As such, any certification authority seeking RTOS robustness testing may discover that it may not be in a separate plan; instead, it may be part of the normal RTOS functional requirements verification.

- The stress testing executed for this operating system did not specifically find any anomalies. However, as part of the planning, preparation, and test case development, an obscure stack overflow condition was discovered as part of the memory model setup and analysis. Accommodation of this remote stack overflow resulted in a slight change in the design architecture, including modification to the MMU model. In particular, placing the stack in the pages of the PowerPC's memory model permits an automatic exception to occur should the stack extend past the page boundary. In this case, a PowerPC feature was used in the memory design architecture to effectively raise an exception that permits the RTOS to handle a potential stack overflow. The important finding here is that for a partitioned-protected system, the RTOS is so highly integrated to the platform and overall system itself that viewing only the RTOS would be inadequate. A clear understanding of the memory model that supports partitioning and associated hardware and firmware design must be coupled with a clear understanding of the RTOS.

- Another concern raised in this application is the lack of the PowerPC's ability to protect the memory BAT registers and page TLB registers from single event upset (SEU). In particular, the PowerPC BATs are not parity-protected, and an SEU could potentially perturb these registers. The applicant's accommodation of this problem is proprietary, but clearly, the PowerPC and similar complex microprocessors require a thorough analysis of the applicant's safety assessments, criticality, software levels, and environment. Similar concerns exist with the Level 1 cache in the PowerPC, because it is also not parity-protected.

- The stress test plan offered in Phase 3 included several tests for stressing the dynamic memory allocation features of the RTOS. In this instance, the dynamic memory allocation feature was removed in order to comply with DO-178B's determinism guidance. This portion of the stress test plan offered in Phase 3 may not be applicable for higher-criticality systems that have eliminated dynamic memory allocation.

  An example of how stress test considerations were part of the overall RTOS design is in the protection of the shared I/O resources in this RTOS. The RTOS has an allocation table that only permits access of certain partition processes to certain I/O drivers. This was built as part of the modified RTOS requirement. Illegal address boundary-type testing on this requirement was used to confirm the requirement, and was already part of the RTOS requirement testing.

- The stress test plan suggested providing a critical resource monitor, a concept offered also in ARINC 653. This RTOS provides no such monitor. It relies upon the hardware MMU for monitoring the space resources. For the timing monitor, it relies upon a combination of the PowerPC decrementer feature and the process checks on their deadline monotonic scheduler to provide coverage. Here again, this monitoring capability was built into the RTOS and having a separate critical resource monitor provided little additional benefit.

- Yet another observation is with the Deadline Monotonic Analysis (DMA) scheduler used in this application. A scheduler can be dynamic or preassigned prior to run time and established on initialization. The latter was done in this applicant's RTOS. The number of processes and calculations of the WCET are critical to the correct operation of the DMA scheduling algorithm. Certification authorities should be aware of the pedigree of any tools used in DMA scheduling and WCET analysis.

- Another consideration is the use of the linker in this partitioned system. Because the RTOS is multipartitioned, a vehicle for specifying what software is used in what partition is needed. For this application, every partition is in a different file at link time, and the linker creates all memory partitions. For highly critical applications, this partitioning allocation mechanism should be carefully scrutinized by any certification authority because the linker generates and integrates code into the target system. It is recommended that some form of validation on the linker from a partition link point of view should be conducted, regardless whether the source is a vendor or an applicant.

6.4 CASE STUDY CONCLUSIONS.

The application of the stress test proposed in Phase 3 to a real project provided some increased level of robustness testing of the RTOS; however, for the subject RTOS under test, it did not discover many new problems or holes in the designed implementation. This was, in part, due to the robustness of the requirements already submitted to the customized RTOS to address partitioning safety concerns. It is likely that a COTS RTOS developed, or even reverse-engineered, under DO-178B guidance would have similar results. This would suggest that the stress test plan could be applied to a DO-178B-ready RTOS, but perhaps it would be more appropriate to an RTOS that was not developed to the DO-178B guidance.

The discovery of note from this exercise is the realization that stress testing needs to be applied beyond the RTOS and into the system with respect to the partitions, timing, monitoring, starvation, queue overflows, and all system application processes.

7. CONCLUSIONS AND RECOMMENDATIONS.

This report studied the hardware and software architectural issues of embedded aviation software systems, with a particular focus on systems with multilevel criticality partitioning provided by some commercial off-the-shelf (COTS) real-time operating systems (RTOS). This report has discussed many of the features that could pose some potential safety concerns for the system under development. The conclusions from this study are shown in the bulleted list below.

As Phase 4 of the investigation into COTS software issues, this report concludes by focusing on related architectural strategies. It was found that there is a desire to reduce computing resources in aviation industry systems and that there is a need to move to integrated modular avionics (IMA) architectures within applicable software applications. As the predominant part of this IMA architecture, it was also found that the RTOS provides services to the system and interacts with highly complex microprocessors to accomplish the goals of the system. Finally, it was

concluded that it logically follows that the complex microprocessors within the RTOS contain features that are controlled by the RTOS and thereby will affect the overall safety of the system.

In addition to the architectural issues of embedded aviation software systems, the Federal Aviation Administration is concerned with the isolation and fault containment provided by the partitioning function within some RTOSs. It was found that memory in partitioned systems must be managed. Also, partitioned RTOS systems are candidates for supporting IMA systems. It is apparent from this research effort that partition-supporting RTOSs require careful requirement, design, and development considerations. Verification of the resulting RTOS and its associated acceptance by the certification authority is difficult, particularly for high-criticality systems.

This report can provide a vehicle for dialog between applicants, COTS vendors, and certification authorities. The research shows that a partition-supporting RTOS is just a part of the overall system's protection mechanism and that further research is needed in the integration of RTOSs, applications, and IMA systems in general. This further research and other recommended research, as well as recommendations, are shown below in the bulleted list of recommendations and further research.

The conclusions from this study are:

- The use of standards such as Portable Operating System Interface and Avionics Application Software Standard Interface by ARINC (ARINC 653) does not mean the RTOS is safe. Meeting ARINC 653 will make the developers include some attributes of protection, yet this says nothing for the approval of such a system within the certification process. Higher-criticality systems will require certain features to provide needed determinism, i.e., deterministic scheduling within a partition, allocation of memory on system start-up, heap, and stack sizes preallocated.

- Board support packages (BSPs) are interfaces between the RTOS and the hardware that initializes the microprocessor and memory devices, and BSPs can also perform low-level cache manipulations. Beyond the understanding of the verification pedigree conducted on the BSP, one should also understand the functions employed during both start-up and normal operations.

- Code developed on a nonpartitioned system may have performed instructions in the Supervisor mode as part of that system's requirements. At times this system attribute can be hidden or masked and reuse of that code into a partitioned system will require a detailed scan of instructions used to determine no Supervisor-mode instructions are employed.

- Historically, most of the functions on an aircraft were federated and although functions may be related, there was enough isolation in the federated architecture that the integration of functions was rather simple. In a partitioned system, functions have easier access to system information (one of the many benefits), but similarly, how the system is built and how the applications interact can be a daunting task. Disciplined approaches to developing and testing large-scale integration efforts are lacking.

- Although single event upset (SEU) was not a specific area of study in this research, complex embedded systems have been suspect to SEUs, particularly at high altitudes. The focus has been on memory where bits can inadvertently flip. Techniques such as error detection and correction (EDAC) are used on this memory to prevent these occurrences, but with complex chips, such as the PowerPC, cache has neither parity nor EDAC capability. Recommendations have been offered to turn off the internal cache, but this eliminates most of the benefit provided by the PowerPC. Alternate methods have been proposed, but these could use further scrutiny. Perhaps most critical in their susceptibility are the memory registers that are used for memory partitioning or machine status.

- Security and safety considerations align themselves fairly well for the most part, and there are initiatives for RTOSs to provide security offerings in their products. Areas of discord between safety and security can exist for both space and time and determinism in general.

- Linkers can combine not only code within a partition but also establish resource allocation tables, executive and kernel code, and establish partition boundaries. This has expanded the authority of the linker and the linker output and resulting target system code should be comprehensively verified.

Recommendations and further research:

- Use of microprocessor features must be understood. The following features should be detailed, perhaps in the Plan for Software Aspects of Certification: when Supervisor/User modes are allowed; when and how the memory models are instantiated; the cache algorithms that are in place; how exceptions and interrupts are handled; and the microprocessor optimization features that are employed.

- Partitioning properties and how they affect partition protection in the space, time, and input/output (I/O) domain should also be understood. Various partition-to-partition communication schemes can exist, and how that communication is managed may compromise safety. Partition schedulers need the authority to terminate partitions that do not relinquish control and handle interpartition timing jitter. Within a partition, task overruns can be handled with a variety of schemes such as ignoring the overrun altogether, task termination, task warm restarts, partition termination, system warm restart, or system reset. These actions can easily affect the safety case for the partitioned system. How the memory management unit (MMU) provides partitioning protection should also be established.

- As with other shared resources, I/O operations should be partitioned or protected in some fashion. Input/Output management is as important as memory and time management on partitioned systems. Different approaches are possible such as kernel-centralized I/O control or partition I/O control itself. Many implementations incorporate some type of task permission table that permits only certain tasks to access specific I/O. A health monitor is then employed to determine if any undesigned accesses are sought by other

tasks or partitions. This safety feature becomes particularly important for systems allowing for hot swapping that can install new functions or upgraded programs to a system.

- Most deterministic schedulers require a worst-case execution time (WCET) so that the process or task can be scheduled in an appropriate time slot. This is particularly difficult for complex microprocessors that use cache and out-of-order execution instructions. Low-criticality level software may use a crude guess or a more structured approach, but certainly as the criticality increases, the WCET estimate requires increased accuracy and validation. In partitioned systems, code and data memory may be effectively partitioned via the MMU; however, on-chip code and data cache does exist that is not partitioned. All partitions can have access to cache when they are running. The initial state of this cache may require cache flushing. Who performs this activity can vary and could impact timing requirements. Another opportunity for cache compromise is where the partition itself could modify cache and leave it in a state that may induce a failure in another partition.

- Health monitor recovery mechanisms can vary and need careful scrutiny. Recovery could be ignore it, suspend the task, kill the task, kill the partition, reset the system, or simply annunciate the fault. These recovery mechanisms can have very serious system effects and, as such, need careful scrutiny.

- Further study is needed on the partition system development domain. Design approaches and constraints, rules for RTOS developers, designers, integrators, and application developers are in need. The complexity of integrating these functions has given rise to more authority and decision making capability by development tools. Partition system building techniques that include building a testable system and effectively using associated tools is necessary.

- Further study is also needed on static memory map analysis techniques and methods for acceptance of deterministic schedulers. Health monitoring capabilities and how the RTOS recovers from detected and identified health problems should be studied.

- Further study is also needed on the implications of a variety of IMA configurations such as partitions on a single central processing unit (CPU), partitions on a distributive system, IMA hybrids (single CPU/distributive), or even IMA systems within IMA systems.

- Several vendors and applicants seek an incremental certification approach with respect to partition-supporting RTOSs, and much more research can be conducted in this area.

8. REFERENCES.

1. J. Krodel, "Commercial-Off-The-Shelf (COTS) Avionics Software Study," FAA report DOT/FAA/AR-01/26, May 2001.

2. R. Thornton, "Review of Pending Guidance and Industry Findings on COTS Electronics in Airborne Systems," FAA report DOT/FAA/AR-01/41, August 2001.

3. V. Halwan and J. Krodel, "Study of Commercial-Off-The-Shelf (COTS) Real-Time Operating Systems (RTOS) in Aviation Applications," FAA report DOT/FAA/AR-02/118, December 2002.

4. Otero, 2002 Internal Presentation on System and Product Values, Pratt & Whitney Aircraft.

5. Fox, Lantner, Marcom, "A Software Development Process for COTS-Based Information System Infrastructure," *Fifth International Symposium on Assessment of Software Tools and Technologies*, June 2-5,1997.

6. Motorola, Inc., "PowerPC Microprocessor Family: The Programming Environments for 32-Bit Microprocessors," Motorola Technical Guide, MPCFPE32B/AD Rev. 1, 1997.

7. Harbour, "A Proposal for the POSIX 1003.13 Revision," The POSIX Real-Time System Services Working Group, April 2002, http://www.opengroup.org/rtforum/rt_os_profiles/uploads/40_1031048941__dot13-revision-slides.pdf.

8. Dorsey, "RTOS Basics," FAA National Software Conference, http://av-info.faa.gov/ software/Conf01/RTOS.pdf, June 2001.

9. N. Zhang, A. Burns, and M. Nicholson, "Pipelined Processors and Worst-Case Execution Time," University of York, UK, 1992.

10. TimeSys Corporation, "The Concise Handbook of Real-Time Systems," 1999.

9. RELATED DOCUMENTATION.

ARINC, "Aviation Application Software Standard Interface, ARINC Specification 653," Airlines Electronic Engineering Committee, 1997.

FAATSO-C153-02, "Integrated Modular Avionics Hardware Elements," DOT/FAA/ACS, May 2002.

Rushby, “Partitioning in Avionics Architectures: Requirements, Mechanisms, and Assurance,” SRI, NASA/CR-1999-209347, June 1999.
http://techreports.larc.nasa.gov/ltrs/PDF/1999/cr/ NASA-99-cr209347.pdf.

DiVito, “A Formal Model of Partitioning for Integrated Modular Avionics,” NASA/CR-1998-208703, August 1998.
http://techreports.larc.nasa.gov/ltrs/PDF/1998/cr/NASA-98-cr208703.pdf.

Bate, Conmy, McDermid, “Generating Evidence for Certification of Modern Processors for Use in Safety-Critical Systems,” Dept. of CS, University of York, York, UK, 2000.

Lundqvist, Stenstrom, “Timing Anomalies in Dynamically Scheduled Microprocessors,” *20th IEEE Real-Time Systems Symposium*, Chalmers Univ. of Tech., Goteborg, Sweden, December 1999.

Hergenhan, Rosenstiel, “Static Timing Analysis of Embedded Software on Advanced Processor Architectures,” Universitat Tubingen, Germany, 1999.

ARINC, “Design Guidance for Integrated Modular Avionics, ARINC Specification 651,” Airlines Electronic Engineering Committee, September 1991.

Smith, “Cache Memories,” *Computing Surveys*, Vol 14, No. 3, September 1982.

Tietz, Leon, “Under the Hood of an Ideal Microkernel,” *COTS Journal*, November 2001, pp. 25-31, 1991.

Jim Alves-Foss, Bob Rinker, and Carol Taylor, “Towards Common Criteria Certification for DO-178B Compliant Airborne Software”
http://www.cs.uidaho.edu/~jimaf/docs/compare02b.htm

Audsley N, Burns A, and Wellings, A., “Implementing a High-Integrity Executive Using Ravenscar,” *Ada Letters,* Vol XXI, No 1, pp. 40-45, 2001.

ARINC 653–Supplement 1, “Avionics Application Software Standard Interface, ARINC specification 653,” September 2003.